Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 93
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 93
Espoo 2004 HUT-TCS-A93

SPECIFICATION-BASED TEST SELECTION IN FORMAL CONFORMANCE TESTING

Tuomo Pyhälä

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 93
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 93
Espoo 2004 HUT-TCS-A93

SPECIFICATION-BASED TEST SELECTION IN FORMAL CONFORMANCE TESTING

Tuomo Pyhälä

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

ISBN 951-22-7240-7
ISSN 1457-7615

Multiprint Oy
Helsinki 2004

ABSTRACT: More complex systems require more efficient quality assurance. Testing is a often used method to achieve this goal. In this work we consider formal conformance testing, which is a field where formally defined conformance between specifications and implementations is studied. In practice this allows the construction of automated testing tools having a solid theoretical foundation and being able to automatically test whether an implementation conforms to its specification.

Test selection refers to the process of trying to select, from a potentially very large set of possible test cases, the tests which most efficiently test the implementation. We consider a set of tests to be efficient, if it tests a large proportion of the implementation behaviour without containing too much redundancy. The proposed test selection method is based on the assumption that the implementation resembles the specification. Therefore, by this assumption, having tested a large proportion of the specification behavior we have also tested a large proportion of the implementation behavior.

To capture our notion of a large proportion of the specification behavior, we formally define a specification-based coverage framework, including several coverage metrics of different levels of granularity. We refine an existing conformance testing algorithm to include the specification coverage based test selection methods based on the framework. The algorithm and the coverage metrics are tied together by a heuristic aiming to increase the coverage metrics. Finally, we implement a tool incorporating the discussed algorithms and make a number of experiments with the tool.

KEYWORDS: formal conformance testing, test selection, on-the-fly testing, specification-based coverage

CONTENTS

1 INTRODUCTION

Nowadays software is present almost everywhere in the technologically advanced society. All the time we use systems containing substantial amount of software. Some examples include cars, lifts, airplanes, telephone- and power-distribution networks, not to even mention large computer systems enabling required efficient data processing in our society.

The reliability of these services requires dependable software. Better methods to produce such software are constantly pursued. As always specifying processes and introducing automation to dull and error prone tasks are ways to improve quality.

One software process model is the *waterfall model* [34]. In the waterfall model software development advances in steps from one phase to another. The phases from the beginning are (i) requirements analysis and definition, (ii) system and software design, (iii) implementation and unit testing, (iv) integration and system testing and (v) operation and maintenance. Of course, software production process is not linear. If we find design flaws in system testing, we have to return to an earlier phase.

The waterfall process will give an overview of activities required in creating new software. There are considerable amounts of testing in this process definition. Therefore producing high quality software will require us to pay close attention to testing activities.

1.1 Software Testing Terminology

Software can be thought consisting of *components*, which are divided to one or more *units*. An unit is a work of a single programmer, maybe several hundred lines of code. When components are integrated, they form bigger components and ultimately the intended *system*, a complete piece of software. [3]

Testing different pieces of software can be divided into *unit testing*, *component testing*, *integration testing* and *system testing*. The concepts were introduced in increasing size of the component being tested. [3]

To hide the complex details of software, programmers introduce interfaces to units and components. Other programmers who build their software on top of these pieces of software may then rely on these interfaces, and do not have to know about the internals. Well defined simple interfaces are a part of good software design. They can also be used as a basis for testing.

In *black box testing*, i.e., *functional testing* the interface of an implementation under test (abbreviated as IUT) is tested [43]. This means that the tester is unable to observe internal details of the IUT and cannot therefore use these observations to make conclusions about the correctness of the implementation. However, tester may take advantage of the specification of the IUT in this task. On the other hand, *white box testing*, i.e., *structural testing* takes advantage of internals of the component[43]. White box testing may include inspections of code and design. Another white box testing approach is to design test cases to execute all statements in the particular software component at least once. Creation of such test cases is impossible with the knowledge of the interface specification only.

When testing software we can identify several subtasks. In the *test generation* phase we generate *test cases*. In the *test execution* phase test cases are executed against an actual implementation. Before this the environment has been prepared in the *test setup* phase, and after execution the results of testing are produced and considered in the *test results evaluation* phase.

This report is about *conformance testing* which we define to be testing whether an implementation conforms to its specification. We will define conformance formally, but for now we can think conformance as behaving in accordance with the specification. Conformance testing is mostly a black-box testing method, although knowledge of the internals of implementation is advantageous.

1.2 Improving Testing Activity

To see the present and future, it is important to survey the past. According to [3] the view of testing has developed as follows.

In the early days of software testing was almost non-existent. At that time there was no difference between debugging and testing. However, software components were small and disposable, in the sense that they were not constantly maintained and improved.

As software became more complicated, the way of thinking changed. Testing was a way to prove that software works. "You see, all test runs work, there are no bugs", a software engineer of that time might have said. However, this claim has no rigid foundation. If a particular test case is successful, it shows that this particular behavior is correct. Practically even small components have tremendous amount of inputs. For example, a routine taking two 32-bit integers as parameters and returning the maximum of two has 2^{64} test cases, which is far too many to be practical. Every substantial software component contains larger components and more complicated interfaces than this simple routine.

After rejecting testing as a way of proving that software works, we could try to prove that software does not work. After all an unsuccessful test case does exactly this. Again, a large number of test cases is a problem, as in practice there always exists one more test case to be run. At a certain point testing has to end so that software may be delivered to customers and users.

Our objective is to stop testing after reaching the required certainty that software works sufficiently well. Running a sensible set of test cases, a *test suite*, whether passing or failing, reduces uncertainty about software reliability. Ultimately we will have ubiquitous testing which will be present in the whole software development process, i.e., development is done in a way that software is easily testable.

One approach to make software more testable, is to utilize automation. Test automation in all subtasks of testing will enable us to run several orders of magnitude more of unique test cases. Increased testing quality will boost substantially our ability to produce software. We can take advantage of new methods in two ways, either increase the quality, or produce larger software systems in the current quality level [3].

1.3 Formal Methods

Formal methods could be described as the use of mathematical reasoning in software development. Traditionally main activities include formal specification, proving properties of specification, constructing programs by mathematically manipulating the specification and verifying program by mathematical argument [18]. Examples of formal specification languages include SDL [24] and Promela [23]. Formal methods have mostly been applied in safety-critical and other systems with high-quality standards, although they are useful in all kind of software systems.

We see following advantages of using formal methods: (i) finding errors early, (ii) making developers think hard the system they are building, (iii) creating more abstract and error proof specifications, (iv) decreasing the cost of development and (v) helping clients understand what they are buying. Slightly different viewpoint can be found in [18].

A flavor of the theory of formal conformance testing which we use in this work has been introduced in [38]. Formal testing promises to be able to automate deriving test cases from formal specifications and to verifying the results of testing automatically.

1.4 Research Background

Testing can be divided on the basis of different subtasks are performed. In the batch-mode testing an explicit test case is generated and then it can be executed one or more times against the implementation. More recent approach is to combine generation and execution of test cases as well as evaluation of the results as one automated and unified process.

Currently there exists several on-the-fly testing tools. Some of them are based on the rigid formal theoretical background and others have been developed using practical experiences and intuition. Real running examples of such on-the-fly testers are TorX [9] and TGV [13], but there are also several other tools.

Existing work on selecting tests and measuring coverage includes, for example, test selection and measuring coverage based on traces [12], test selection using test purposes [7], using coverage information to find bugs faster in model checking Java programs [17], using heuristic methods for test selection [30], using bounded model checking in test sequence generation [44], and using coverage criteria in test generation for white box testing [37].

To make such testing more feasible we need measures to make test selection more intelligent. The existing on-the-fly testers have achieved very positive results by just randomly sending valid inputs to the implementation and checking whether the outputs are allowed according the specification. We call this “poking around”, which is effective because an automated tool is able to do this several decades faster than a human as humans lack the precision and stamina generate all possible complicated tests nor the generation speed of the testing tool. However, an automated testing tool has the disadvantage of executing too many similar tests, if they are generated randomly. We address this disadvantage with test selection methods based on specification coverage.

1.5 Contributions of this Report

The contribution of this report is the design of formally defined specification coverage based test selection methods for an on-the-fly testing tool, and the implementation of this tool. Our design is based on the idea that we know the specification, but the details of the implementation are hidden, although we assume that the implementation resembles specification. Moreover, we assume, that by having tested a large proportion of the specification behavior we have tested also a large proportion of the implementation behavior. To capture our notion of a large proportion of the specification behavior, we define formally a framework for specification coverage including coverage metrics on different levels of granularity. The framework contains the general terminology and desirable properties of the coverage metrics.

We also refine the algorithm from [41] to include methods for test selection. We do this by introducing hooks for heuristics. Then we combine this to the a coverage metric – Petri net transition coverage – by constructing a heuristic to increase this coverage using either greedy search or bounded model checking. We also adapt some existing terminology to analyze on-the-fly testing algorithms. Specifically concepts of soundness, exhaustiveness and completeness are defined to be similar as those for test suites in [39] and related concept called omnipotentness is introduced.

To evaluate our tool in practice, we experiment with two different problems. One case is a conference protocol [11] which has been used to test other testing tools [4]. Another case is a combinatoric lock which we suppose to be hard to test with random testing.

1.6 Structure of the Report

The structure of the report is as follows. Chapter 2 will introduce necessary formalisms for conformance testing. In Chapter 3 we consider the attachment of tester to the implementation and different approaches to test execution and generation, should they be combined as in an on-the-fly tester or not. Chapter 4 is to introduce a coverage concept and present our framework for coverage metrics for test selection. Chapter 5 introduces algorithms for testing which we extend with coverage based heuristics. We also discuss soundness and completeness of extensions to the algorithms. Chapter 6 reviews our implementation. In Chapter 7 we represent the results of the experiments with our implementation. Chapter 8 draws the conclusions of this work.

2 FORMALISMS FOR CONFORMANCE TESTING

In conformance testing we define conformance as a relation between implementations and specifications. To treat these formally, we need formalisms to represent them. In this section we introduce labelled transition systems – a formalism to represent specifications and implementations – and define the **ioco** conformance relation using this formalism. However, a labelled transition system is a low level formalism in the sense that specifications in such a formalism are very large and at low abstraction level. We have chosen labelled transition systems as low level formalism, because they are well known, there exists rigid theory of conformance testing for them and they are able to specify practically any system.

There are lots of other formalisms, which have a higher abstraction level and hence are more understandable. Such formalisms make the specification easier as the specifications are in more compact form. They also induce corresponding labelled transition systems specifying the same behavior. In our tool we use 1-safe Petri nets [10], which are a bit higher level formalism, although still straightforward to implement using simple algorithms. Besides there are tools to create 1-safe Petri nets from high level nets having even higher abstraction level. Therefore, it seems reasonable to use 1-safe Petri nets which can be easily implemented and having existing tools giving substantial additional functionality in the form of higher level specifications.

2.1 Labelled Transition Systems

Labelled transition systems (abbreviated as LTS) are a well known formalism to specify system behavior. A labelled transition system

consists of states and labelled transitions between states.

Definition 2.1. *A* labelled transition system *is a four tuple* (S, L, Δ, s_0), *where* S *is set of states,* L *is a finite set of* labels *with a special symbol* $\tau \notin L$, $\Delta \subseteq S \times (L \cup \{\tau\}) \times S$ *is the transition relation, and* s_0 *is the* initial state.

When testing systems we have visible events and hidden internal actions. We define visible events as follows.

Definition 2.2. *A transition* $t = (s, a, s')$ *of some LTS* $p = (S, L, \Delta, s_0)$ *is a* visible transition *if* $a \in L$. *The set of all visible transitions in the set of transitions* Δ *is denoted as* $\Delta_v \subseteq \Delta$

In connection with LTS's we use the notation L^*, where L is some set of symbols, to denote concatenation of these symbols, i.e., $\sigma = a_1 \cdot a_2 \cdot a_3 \cdots \in L^*$, iff $\forall i \leq |\sigma| : a_i \in L$.

Next we introduce a notation for labelled transition systems. The notation used here is similar to that of [9].

Definition 2.3. *With* $\sigma \setminus a$ *we denote that we remove symbol* a *from a trace* $\sigma \in L^*$. *This is defined inductively as follows. For* $\sigma = \epsilon$, $\sigma \setminus a$ *for any* a *is* ϵ. *Otherwise, let* $\sigma = a_1 \cdot \sigma'$, *then the resulting trace* $\sigma \setminus a$ *equals* $\sigma' \setminus a$ *if* $a_1 = a$ *and* $a_1 \cdot (\sigma' \setminus a)$ *if* $a_1 \neq a$.

Definition 2.4. *We define the following for LTS $p = (S, L, \Delta, s_0)$. In the following definitions $s, s' \in S$, $\mu, \mu_i \in L \cup \{\tau\}$, $a, a_i \in L, \sigma \in L^*$:*

$$\begin{aligned}
s \xrightarrow{\mu} s' &=_{def} (s, \mu, s') \in \Delta, \\
s \xrightarrow{\mu_1 \ldots \mu_n} s' &=_{def} \exists s_0, s_1, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \cdots \xrightarrow{\mu_n} s_n = s', \\
s \xrightarrow{\mu_1 \ldots \mu_n} &=_{def} \exists s' : s \xrightarrow{\mu_1 \ldots \mu_n} s', \\
s \not\xrightarrow{\mu_1 \ldots \mu_n} &=_{def} \neg(s \xrightarrow{\mu_1 \ldots \mu_n}), \\
s \stackrel{\epsilon}{\Longrightarrow} s' &=_{def} s = s' \vee s \xrightarrow{\tau \ldots \tau} s', \\
s \stackrel{a}{\Longrightarrow} s' &=_{def} \exists s_1, s_2 : s \stackrel{\epsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \stackrel{\epsilon}{\Longrightarrow} s', \\
s \stackrel{a_1 \ldots a_n}{\Longrightarrow} s' &=_{def} \exists s_0 \ldots s_n : s = s_0 \stackrel{a_1}{\Longrightarrow} s_1 \stackrel{a_2}{\Longrightarrow} \cdots \stackrel{a_n}{\Longrightarrow} s_n = s', \\
s \stackrel{\sigma}{\Longrightarrow} &=_{def} \exists s' : s \stackrel{\sigma}{\Longrightarrow} s', \\
s \stackrel{\sigma}{\not\Longrightarrow} &=_{def} \neg(s \stackrel{\sigma}{\Longrightarrow}), \\
\mathit{traces}(s) &=_{def} \{\sigma \in L^* \mid s \stackrel{\sigma}{\Longrightarrow}\}, \\
\mathit{init}(s) &=_{def} \{\mu \in L \cup \{\tau\} \mid s \xrightarrow{\mu}\}, \\
s\ \textbf{after}\ \sigma &=_{def} \{s' \mid s \stackrel{\sigma}{\Longrightarrow} s'\}, \text{ and} \\
\mathit{vis}(\sigma \in (L \cup \{\tau\})^*) &=_{def} \sigma \setminus \tau.
\end{aligned}$$

To extend previous definitions to state sets we define following:

Definition 2.5. *Let $p = (S, L, \Delta, s_0)$ be an LTS. In the following definitions $S' \subseteq S$, $\mu, \mu_i \in L \cup \{\tau\}$, $a, a_i \in L$:*

$$\begin{aligned}
\mathit{init}(S') &=_{def} \bigcup_{s \in S'} \mathit{init}(s), \\
S'\ \textbf{after}\ \sigma &=_{def} \bigcup_{s \in S'} s\ \textbf{after}\ \sigma
\end{aligned}$$

We also overload previous notations in following way. Often when considering the conformance, we consider what happens to an LTS in initial state after a sequence of events. Therefore in the sake of simplicity we may use LTS instead of it's initial state. So whenever reader sees LTS in the place of state, it denotes the initial state of the LTS.

Example 2.1. We use intuitive graphical notation for LTS's, where states are denoted by circles and transitions with arrows with the corresponding label on the side. Consider as an example LTS in Fig. 1. Let us denote this as the p. Let the set of labels be $L = L_I \cup L_U$, where $L_I = \{a\}$ and $L_U = \{b, c\}$ respectively. Later these sets are called the set of input labels and the set of output labels. Then, for instance, p **after** $a = \{s_1, s_3\}$ and $s_0 \xrightarrow{a} s_1$, $s_0 \not\xrightarrow{a} s_3$, $s_0 \stackrel{a}{\Longrightarrow} s_3$.

Definition 2.6. *A divergence free, or strongly convergent, LTS does not contain an infinite execution of τ transitions, i.e., $\exists n \in \mathbb{N} : \forall s_1, \ldots, s_n$: if $s_1 \xrightarrow{\quad\tau\quad} s_1 \cdots s_{n-2} \xrightarrow{\quad\tau\quad} s_{n-1}$ then $s_{n-1} \not\xrightarrow{\quad\tau\quad} s_n$.*

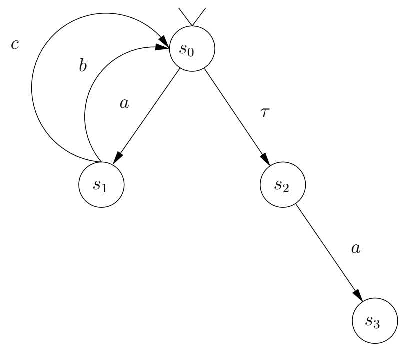

FIGURE 1: Our running example p

Definition 2.7. *An LTS* $p = (S, L, \Delta, s_0)$ *has finite behavior if* $\forall\ \sigma \in L^*$ *such that* $s_0 \stackrel{\sigma}{\Longrightarrow}$ *it holds that* $|\sigma| < n$ *for some fixed finite n.*

In this work, we restrict ourselves to divergence free labelled transition systems, which are denoted as $\mathcal{LTS}$, with $\mathcal{LTS}(L)$ we denote labelled transition systems with label set L.

Our model of testing distinguishes between inputs and outputs, to be more suitable for testing systems which have distinct inputs and outputs. To achieve this, we define a subclass of labelled transition systems called *input-output transition systems (IOTS)*. We also require that IOTS must always accept all inputs, therefore we don't have to consider a situation where *implementation under test* (abbreviated as IUT) is given input which is not enabled.

Definition 2.8. *An* input-output transition system *(abbreviated as IOTS) is a LTS with the following restrictions. The set of labels L is divided into input labels* L_I *and output labels* L_U*, such that* $L = L_I \cup L_U$ *and* $L_I \cap L_U = \emptyset$. *Furthermore, for an IOTS it is required that* $\forall s \in S : \forall a \in L_I : s \stackrel{a}{\Longrightarrow}$. *The latter restriction forces that an IOTS must always accept all input labels. We denote the set of all input-output transition systems with* $\mathcal{IOTS}$.

2.2 Conformance Testing

Conformance testing aims to show that a particular implementation conforms to its specification. This can be done by testing the implementation. The problem is to generate, select and execute test cases and to verify their results. Conformance testing is particularly useful in testing implementations of communication protocols. In tele- and data communication field the products of different vendors must be able to cooperate, and this can achieved by conforming to a common specification.

Formal conformance testing formalize the concepts of conformance testing [38]. Essential notions include the implementation, specification and conformance relation between these two. For treating implementations and specifications, the formalisms introduced previously in this chapter are useful. Conformance relation is relation between conforming implementations and their specifications. If an implementation conforms to a specification, it is in this relation with the specification. Therefore the relation defines exactly the conformance.

2.3 Conformance Relation **ioco**

The basic idea of conformance relation **ioco** is to model conformance in a way that when testing with behavior found in the specification, the implementation acts in the same way. We define **ioco** for labelled transition systems [9].

Often real systems have states, where no output can be observed. To take account in our formal theory the absence of outputs, the *quiescence* concept has been introduced and the **ioco** conformance relation includes this concept. Intuitively, it is defined that a state is quiescent if and only if only input transitions exist in the state. No state with output or internal transitions is quiescent.

Definition 2.9. *A state s of $i \in \mathcal{LTS}$ with labelling $L = L_I \cup L_U$, $L_I \cup L_U = \emptyset$ is quiescent iff $\forall \mu \in L_U \cup \{\tau\} : s \overset{\mu}{\not\longrightarrow}$. We define quiescence function $\delta(s)$ such that, if s is quiescent then $\delta(s) =$ true and otherwise $\delta(s) =$ false.*

Example 2.2. In our running example quiescent states are s_2 and s_3.

The formal concept of out sets is introduced to represent the possible outputs of some particular state.

Definition 2.10. *Let s be a state of $i \in \mathcal{LTS}$ with labelling $L = L_I \cup L_U, L_I \cup L_U = \emptyset$, then*

$$out(s) \quad =_{def} \quad \{a \in L_U \mid s \overset{a}{\Longrightarrow}\} \cup \{\delta \mid \delta(s)\}.$$

Definition 2.11. *Let $S' \subset S$ be a set of states of $i = (S, L_I \cup L_U, \Delta, s_0) \in \mathcal{LTS}$ ($L_I \cup L_U = \emptyset$), then*

$$out(S') \quad =_{def} \quad \bigcup_{s \in S'} out(s).$$

Example 2.3. In our running example, the out set of state s_0 $out(s_0) = \emptyset$ and $out(\{s_0, s_2\}) = \{\delta\}$.

Suspension traces are the traces including suspensions, i.e., apart from having symbols from the label set L, they also have δ symbols to denote suspensions.

Definition 2.12. *The notation $s \xrightarrow{\mu} s'$ is extended to include the suspensions in following way. If $\delta(s)$, then $s \xrightarrow{\delta} s$. The $s \overset{a}{\Longrightarrow} s'$ as well as other notations in Def. 2.4 are extended to use the redefined $s \xrightarrow{\mu} s'$ as basis for them. The set of suspension traces is defined as*

$$Straces(s) \quad =_{def} \quad \{\sigma \in \{L \cup \{\delta\}\}^* \mid s \overset{\sigma}{\Longrightarrow}\}.$$

Example 2.4. The suspension trace $ababa\delta \in Straces(p)$, where p is our running example. For instance, $a\delta ba \notin Straces(p)$ is not a suspension trace of our running example.

Suspension free trace is a trace which does not contain any suspensions. A suspension free trace can be obtained from suspension trace by removing suspension [1].

Definition 2.13. *If σ' is a suspension trace, then the corresponding suspension free trace is obtained by $\sigma = \sigma' \setminus \delta$.*

Example 2.5. Consider suspension trace $\sigma = ababa\delta$. The corresponding suspension free trace $\sigma' = \sigma \setminus \delta = ababa$.

Now we have introduced necessary concepts to formally define the **ioco** conformance relation. Intuitively it models that some specific implementation conforms to corresponding specification. Formal definition follows [39]:

Definition 2.14. *Let $i \in \mathcal{IOTS}$ be an implementation and $s \in \mathcal{IOTS}$ be a specification, then*

$$i \ \mathbf{ioco} \ s \quad =_{def} \quad \forall \sigma \in \mathit{Straces}(s) : \mathit{out}(i \ \mathbf{after} \ \sigma) \subseteq \mathit{out}(s \ \mathbf{after} \ \sigma).$$

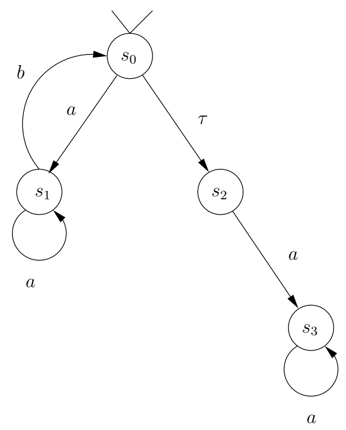

FIGURE 2: An example of an implementation p_1'

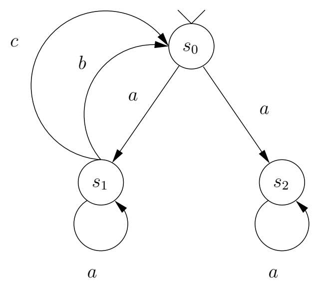

FIGURE 3: An example of an implementation p_2'

[1]Taking suspension free trace of a trace containing no suspensions is a no-op.

Example 2.6. Consider conformance in the light of an example. In Fig. 2 there is an implementation IOTS p'_1. Is this IOTS p'_1 in **ioco** relation with the specification p, where p is our running example? Obviously, implementation p'_1 has less functionality than specification p, it can't ever output c. However, it is conforming, so p'_1 **ioco** p. If you look at the definition, this is due that $out(p'_1 \textbf{ after } a) = \{b, \delta\}$ and $out(p \textbf{ after } a) = \{c, b, \delta\}$ and therefore $out(p'_1 \textbf{ after } a) \subseteq out(p \textbf{ after } a)$.
Another example is p'_2 in Fig. 3. Is this **ioco** conforming w.r.t. specification p? Lets consider $\delta a \in Straces(p)$. We find out the following out sets $out(p \textbf{ after } \delta a) = \{\delta\}$ and $out(p'_2 \textbf{ after } \delta a) = \{b, c, \delta\}$. Therefore $out(p'_2 \textbf{ after } \delta a) \not\subseteq out(p \textbf{ after } \delta a)$ and p'_2 $\cancel{\textbf{ioco}}$ p.

2.4 Suspension Automata

A suspension automaton is a more explicit representation of suspension traces. In a conventional specification LTS we do not have explicit suspension arcs, and the LTS may be also non-deterministic. A suspension automaton is constructed by making suspensions explicitly visible and determinizing the result. We define *suspension automaton* (SA) as follows [39].

Definition 2.15. *Let $p = (S, L, \Delta, s_0)$ be a specification LTS (with the requirements $L = L_U \cup L_I$ and $L_I \cap L_U = \emptyset$) , then corresponding suspension automaton is the LTS $SA(p) = (S^{sa}, L \cup \{\delta\}, \Delta^{sa}, {s_0}^{sa})$, where*

$$\begin{aligned}
S^{sa} &=_{def} \mathcal{P}(S) \setminus \emptyset, \\
\Delta^{sa} &=_{def} \{q \xrightarrow{a \in L} q' | q \in S^{sa}, q' = \{s' \in S | \exists s \in q : s \xRightarrow{a} s'\} \in S^{sa}\} \cup \\
&\qquad \{q \xrightarrow{\delta} q' | q, q' \in S^{sa} \text{ and } q' = \{s \in q | \delta(s)\}\}, \text{ and} \\
{s_0}^{sa} &= \{s \in S | s_0 \xRightarrow{\epsilon} s\}.
\end{aligned}$$

The construction used here is similar to the determinization of a finite automaton [29], except for adding the suspensions. The suspension automata itself is of course also an LTS. However, it is different from the original LTS, as it specifies all the suspension traces explicitly in a deterministic manner. The suspension automata has also potentially exponentially more states reachable from the initial states than the original specification LTS, as the states of the suspension automata are the power set of the states of the original LTS.

Example 2.7. The suspension automaton of our running example is such as in Fig. 4. States correspond to state sets of original suspension automaton as follows $S_0 = \{s_0, s_2\}$, $S_1 = \{s_1, s_3\}$, $S_2 = \{s_2\}$, $S_3 = \{s_3\}$. There are also other non reachable states resulting from Cartesian product used in the definition, however, these are not listed here nor in the figure. Initial state of the suspension automaton is S_0.

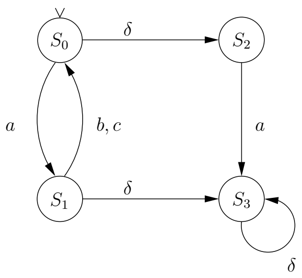

FIGURE 4: The Suspension automaton of our running example

2.5 1-safe Petri Nets

Petri nets [10] are a formalism which enables a more compact representation of systems than simple labelled transition systems as introduced in Def. 2.1. A similar compact representation could also be achieved using synchronizations of such simple labelled transition systems, but for this work we chose labelled 1-safe Petri nets.

Petri nets which are 1-safe are a simpler formalism than the general Petri nets. They are guaranteed to induce LTS's with a finite set of states, moreover algorithms processing them are easier to implement. As the representation of markings is guaranteed to be finite, the implementation does not need to be able to grow data structures representing arbitrarily large markings. Also the finite number of markings (states) guarantees, that we do not have to deal with infinite sets. Therefore we felt that we can faster approach our goal, if we do not spend so much time implementing our formalism and we chose 1-safe Petri nets for the task.

A three tuple $N = (P, T, F)$ is a *net*, where P and T are finite sets of *places* and *transitions*, respectively. The place and transition sets are distinct, i.e., $P \cap T = \emptyset$. The *flow relation* is $F \subseteq (P \times T) \cup (T \times P)$. Transitions and places can also be called *nodes*. By $F(x, y)$ we denote the *characteristic function* of F. It is defined as if $(x, y) \in F$ then $F(x, y) = 1$ and if $(x, y) \notin F$ then $F(x, y) = 0$.

We use Fig. 5 as our running example of Petri nets during this section. It is a graphical representation of a net. Places are represented with circles, transitions with rectangles. The flow relation is represented with arrows. If $(x, y) \in F$, then there is arrow starting from x and pointing to y. The additional features in the figure are to be represented later in this chapter.

Preset of a node a is ${}^\bullet a =_{def} \{x \mid (x, a) \in F\}$. Similarly, *postset* of a node a is $a^\bullet =_{def} \{x \mid (a, x) \in F\}$. In our running example $\mathrm{T6}^\bullet = \{\mathrm{P3}, \mathrm{P6}\}$ and ${}^\bullet\mathrm{P6} = \{\mathrm{T5}, \mathrm{T6}\}$.

A *marking* of a net (P, T, F) is a function $M : P \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers including zero. Marking associates a number of tokens with each place.

A four tuple $N = (P, T, F, M_0)$ is a *Petri net* if (P, T, F) is a net, and M_0 is a marking of this net. A transition t is *enabled* in a marking M, iff $\forall p \in {}^\bullet t : M(p) \geq F(p, t)$. If transition t is enabled in a marking M, it can occur leading to marking M', where $\forall p \in P : M'(p) = M(p) - F(p, t) + F(t, p)$. This is denoted by $M \xrightarrow{t} M'$. In our running example, there is an initial

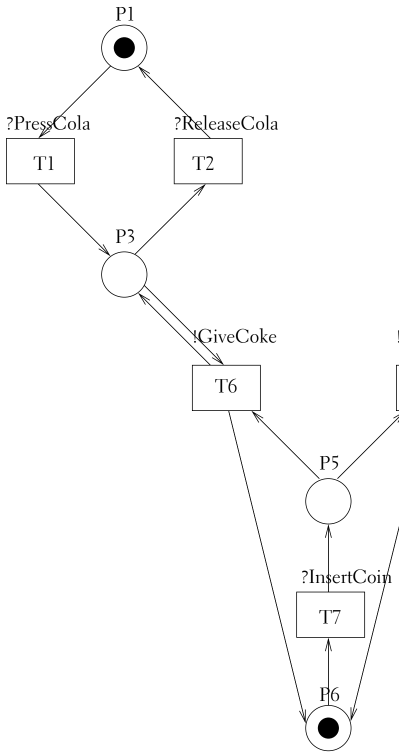

FIGURE 5: an Example of a Labelled Petri net

marking marked with black filled circles inside the places. In the initial marking the transitions T1, T4 and T7 are enabled.

In further discussion we will discriminate between *reachable markings* and all markings. There are infinitely many markings for most Petri nets, but however, not all of these are reachable. In practice only markings reachable from the initial marking affect the specified behavior and therefore no other markings are significant from our point of view. We wish to analyze only whether observed behavior corresponds to the specified behavior.

Definition 2.16. *Let $N = (P, T, F, M_0)$ be a Petri net. The set of reachable markings RM(N) is defined to be the smallest set fulfilling following two conditions:*

(i) $M_0 \in RM(N)$*, and*

(ii) If $M \in RM(N)$ *and* $M \xrightarrow{t} M'$ *then* $M' \in RM(N)$

A marking M is reachable, iff $M \in RM(N)$.

To model real systems, we wish to give some meaning to transitions. We do this by defining a labelling on transitions.

Definition 2.17. A labelled net *is a four tuple* (P, T, F, λ)*, where* (P, T, F) *is a net, and* $\lambda : T \to L \cup \{\tau\}$ *attaches labels to transitions from finite set of labels* L.

In our running example the λ is given by labels beside the transitions, for example λ(T1) = "?PressCola" and λ(T5) = "!GiveSoda".

To use our **ioco** conformance relation defined for LTS's we have to define how a LTS is formed on the basis of a Petri net. We include the reachable markings as the states of such an LTS and if some transition exists between the states then we include an arc with the same label to the LTS. We define an unfolding of a labelled Petri net to a LTS as follows

Definition 2.18. *A labelled Petri net* $N = (P, T, F, \lambda, M_0)$ *induces a labelled transition system defined as* $LTS(N) = (S, L, \Delta, s_0)$*, where* $S = RM(N)$*,* L *is the same set of labels as in the corresponding labelled net,* $\Delta = \{(M, l, M') \mid \exists t \in T : M \xrightarrow{t} M' \wedge \lambda(t) = l\}$*, and* $s_0 = M_0$*.*

A marking which contains at most one token in each place is called *1-safe marking* [10]. Formally marking M of net (P, T, F) is 1-safe, iff $\forall p \in P : M(p) \leq 1$. A *1-safe Petri net* is a Petri net where all its reachable markings are 1-safe. The example Petri net in Fig. 5 is 1-safe.

In testing context it is important to discriminate between visible events and internal actions. To capture these with our labelled Petri nets we define as follows

Definition 2.19. Visible labels *for labelled Petri net* $N = (P, T, F, \lambda)$ *are those in* L*, where* L *is the set of labels. The* τ *is an internal action.*

Definition 2.20. Visible transitions *of labelled Petri net* $N = (P, T, F, \lambda)$ *are transitions* $t \in T$ *such that* $\lambda(t) \neq \tau$*. We denote with* $vistrans(N)$ *the set of visible transitions of* N*.*

3 INTRODUCTION TO CONFORMANCE TESTING

Testing the implementation under test (IUT), contains several subtasks: generating tests, executing tests and evaluating the results are the main subtasks. In this section we will introduce two ways of testing: batch mode testing and on-the-fly testing. These ways organize the subtasks differently. At first, however, we will consider how to attach a tester to the implementation.

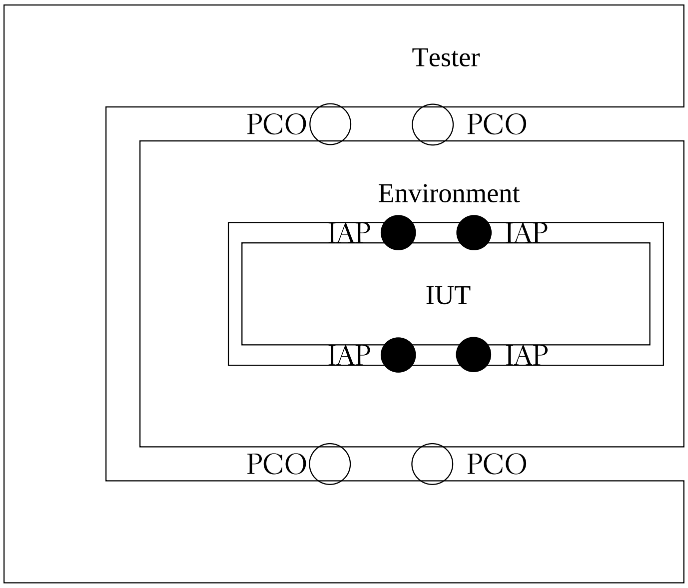

FIGURE 6: Overview of the test setup

Our view of testing can be seen in the Fig. 6. A tester, being either some automated tool or a human, drives the testing process. *Point of control and observation* (abbreviated as PCO) is an attachment point between the tester and the environment. On the other hand, the IUT communicates with the environment through *implementation access points*, or IAPs. A similar view of the testing has been presented in [31]. The IUT and environment together form the SUT, system under testing.

The distinction between IAPs and PCOs is made, because quite often a tester cannot directly attach to an implementation. There might be protocol layers, operating system or some other environment between the tester and an actual implementation under test and therefore the communication is indirect. Thus the tester observes and produces events at PCOs. These facts affect testing. Buffers, delays and other effects introduced by the environment have to be considered and somehow dealt with when testing the IUT.

The theory introduced so far is based on the idea of synchronous testing. The specification and implementation are synchronized and events happen instantaneously in both. There is no notion of communication, delays or buffers between implementation and tester. The tester should be just a synchronized specification tightly bound to an implementation. This model, although formalizes nicely, is not applicable in most situations arising in practice due to missing parts modeling important subset of the real IUT.

There are situations in the real world where asynchronous testing is required. Asynchronous testing includes features lacking from the synchronous testing. However, formal treatment of asynchronous testing is somewhat tricky, for an example of an approach see [38]. In the scope of this work we won't go through it formally.

We use the synchronous testing theory, and try to include the environment to our model in a way that we can synchronously attach to it at PCO's. This requires us to include buffers, delays and non-determinism in the specification. For example the order of outputs observed may vary and in fact all the orders are correct. This is handled by non-determinism. Some inputs may remain unprocessed inside the environment for some time and the IUT may do something else at the same time. This is handled by the buffers.

This approach isn't however easy. Modeling is not straightforward, because often the environment (protocol stack, operating system etc.) is provided by some third party and no exact knowledge of implementational details is available to practitioners of testing. And from the theoretical point of view, for example infinite queues are used [38].

3.1 Batch-mode Testing

In batch mode testing the tester has previously created *test cases*, which it executes (a set of test cases may be thought as a "batch" in contrast to on-the-fly testing, where they are derived on-the-fly). Such a test case defines how the tester (in batch-mode testing) proceeds. It can be seen as a LTS with following restrictions [39].

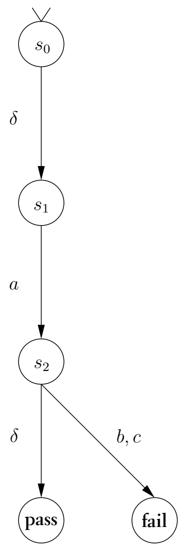

FIGURE 7: Test case t_p for the running example

Definition 3.21.
A test case t *is a labelled transition system* $(S, L_I \cup L_U \cup \{\delta\}, T, s_0)$ *such that*

1. *t is deterministic and has finite behavior,*
2. *S contains the terminal states* **pass** *and* **fail***, with init(***pass***) = init(***fail***) = $\emptyset$, and*
3. *for any state $s \in S$ of the test case one the following is true: (i) $s \in \{\mathbf{fail}, \mathbf{pass}\}$ or (ii) init$(s) = \{a\}$ for some $a \in L_I$ or (iii) init$(s) = L_U \cup \{\delta\}$.*

The class of test cases over L_U *and* L_I *is denoted as* $\mathcal{TEST}(L_U, L_I)$.

These test cases are deterministic and therefore completely define tester behavior in the sense that once the tester is executing a particular test case, it doesn't have to make any choices as it can follow the "instructions" encoded in the test case. An implementation may pass or fail particular test case without any notion of conformance relation or specification. Intuitively synchronized execution of a passing implementation and its specification does not lead to fail state. We will define this more precisely soon.

Example 3.8. In Fig. 3.1 we have an example of test case t_p. It has been created for our running example and detects the faulty implementation in Fig. 3, because when the test case and implementation are synchronized they have a trace leading to fail state. We will explain this in detail after defining properly when test case passes and when it fails.

A *test suite* is a collection of test cases. Formally, a test suite T is $T \subseteq \mathcal{TEST}(L_U, L_I)$.

Sometimes we need to synchronize LTS's, for example to implementation under testing and specification. To accomplish this, we define the following [2] (definition is very similar to that of [39].).

[2] The intended use is $t]|i$, t being the test case and i being the implementation.

Definition 3.22. *The operator* $\rceil| : \mathcal{LTS}(L_I \cup L_U \cup \{\delta\}) \times \mathcal{IOTS}(L_I, L_U) \rightarrow \mathcal{LTS}(L_I \cup L_U \cup \{\delta\})$ *produces the synchronization of LTS's. Let* $p = (S^p, L, \Delta^p, {s_0}^p) \in \mathcal{LTS}$ *and* $u = (S^u, L, \Delta^u, {s_0}^u) \in \mathcal{IOTS}$*, then the synchronization of* p *and* u*,* $p\rceil|u = (S, L, \Delta, s_0) \in \mathcal{LTS}$*, is defined as* $S = S^p \times S^u$*,* L *is the common set of labels,* Δ *is the smallest relation fulfilling all of the following:*

$$
\begin{aligned}
\forall s, s' \in S^p : (s \xrightarrow{\tau} s' \quad & \textit{implies} \quad \forall x \in S^u : ((s, x), \tau, (s', x)) \in \Delta),\\
\forall x, x' \in S^u : (x \xrightarrow{\tau} x' \quad & \textit{implies} \quad \forall s \in S^p : ((s, x), \tau, (s, x')) \in \Delta),\\
\forall s, s' \in S^p : \forall x, x' \in S^u : & \\
\forall a \in L_I \cup L_U : & \\
(s \xrightarrow{a} s' \text{ and } x \xrightarrow{a} x' \quad & \textit{implies} \quad ((s, x), a, (s', x')) \in \Delta),\\
\forall s, s' \in S^p : \forall x \in S^u : & \\
s \xrightarrow{\delta} s', s \not\xrightarrow{\tau}, x \not\xrightarrow{\tau} & \\
\text{and } \forall a \in L_U : x \not\xrightarrow{a} \quad & \textit{implies} \quad ((s, x), \delta, (s', x)) \in \Delta),
\end{aligned}
$$

and $s_0 = ({s_0}^p, {s_0}^u)$.

Now we formally define a test trace and test execution according [39].

Definition 3.23.

1. *A* test trace *of a test case* $t \in \mathcal{TEST}(L_U, L_I)$ *with an implementation* $i = (S^i, L_I \cup L_U, \Delta^i, s_0^i \in \mathcal{IOTS}(L_I, L_U)$ *is a trace* $\sigma \in$ traces$(t\rceil|i)$ *of the synchronous parallel composition of* t *and* i *iff* $\exists i' \in S^i : (t, i) \stackrel{\sigma}{\Longrightarrow} (\mathbf{pass}, i')$ or $(t, i) \stackrel{\sigma}{\Longrightarrow} (\mathbf{fail}, i')$.
2. *An implementation* i passes *a test case* t *iff all their test traces lead to the* **pass***-state of* t*:*

   i **passes** $t =_{def} \forall \sigma \in$ traces$(s\rceil|i), \forall i' : (t, i) \stackrel{\sigma}{\not\Longrightarrow} (\mathbf{fail}, i')$.
3. *An implementation* i passes *a test suite* T *iff it passes all test cases in* T*:* i **passes** $T =_{def} \forall t \in T : i$ **passes** t.
4. *An implementation fails a test suite* T *iff it does not pass it.*

Example 3.9. Recall test case t_p for our running example p in Fig. 3.1 and faulty implementation p_2' in Fig. 3. The synchronization $t_p\rceil|p_2'$ is in Fig. 8. The implementation does not pass this test case because there exists the path $(s_0, s_0) \xrightarrow{\delta} (s_1, s_0) \xrightarrow{a} (s_2, s_1) \xrightarrow{b} (\mathbf{fail}, s_0)$ and therefore a test trace δab, which does not lead to a **pass** state of the test case.

An *exhaustive test suite* will not pass with non-conforming implementation, i.e., if exhaustive test suite terminates with verdict pass, then the implementation is conforming. A *sound test suite* is a test suite, which will not fail on conforming implementation. A *complete test suite* is sound and exhaustive [39].

As an interesting note, Def. 3.23 works also with a suspension automaton based *explicit tester*. Such a tester is constructed by adding from each state of the suspension automaton arc to the **fail**-state state with each disabled label,

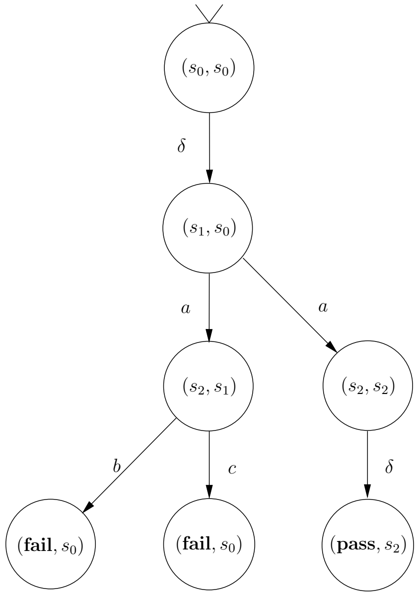

FIGURE 8: Example of test case and implementation synchronized

i.e., if S is the set of the states in suspension automaton, $\forall s \in S$ add an arc with label $L_U \setminus out(s)$ to the **fail**-state. Such an explicit tester can also be used in the place of test case t in Def. 3.23. We claim that this is a sound and complete method, as all possible failing traces lead the **fail**-state in the tester.

The definitions here are theoretical. They are based on the assumption that implementations are formal objects, not real hard- and software. Unfortunately, in practice the implementation under testing is very likely to be a non-formal object. For such object these definitions do not directly apply.

For example one may not be able to obtain all the traces of the implementation. A non-deterministic implementation may never select a possible choice, and therefore all the obtainable test runs pass although the implementation is non-conforming. We consider this a hard problem, and avoid detailed discussion of it by assuming that implementation lets us detect the faults. By this we mean, that the implementation does not constantly keep showing conforming behavior to the tester in the case that the trace leading to non conforming behavior has been executed.

We represent also definitions for these terms when used in context of a non-formal implementation. To model the communication with the IUT we introduce some notation. In the following i is the implementation under test. These concepts are abstractions of actual events at some of the PCOs. Actual events vary from implementation to implementation, but we remain on an abstract level to avoid details. By Stimulate(i, a) we mean that, the implementation i is given input a, by ObserveOutput(x) we mean that the output x has been observed from the implementation or that a timeout has occurred, if $x = \delta$. Timeout means that no output was observed within a time bound. The exact time bound is an implementation dependent parameter.

Running a test case can be done as in Algorithm 3.10 [9].

Algorithm 3.10. Let $p = (S, L_I \cup L_U \cup \{\delta\}, \Delta, s_0) \in \mathcal{TEST}(L_U, L_I)$ be the test case to be run

procedure ExecuteTest(p, i) {
  $s := s_0$;
  while ($s \notin \{\textbf{pass}, \textbf{fail}\}$) {
    if ($\exists a \in \text{init}(s) : a \in L_I$) {
      Stimulate(i,a);
    }
    else {
      // OBSERVEOUTPUT RETURNS EITHER OUTPUT OBSERVED ($x \in L_U$)
      // OR TIMEOUT (δ)
      x := *ObserveOutput*(i);
      // UPDATE THE CURRENT STATE ACCORDING
      // TO THE OBSERVED OUTPUT
      $s := s \xrightarrow{x}$;
    }
  }
  if ($s = \textbf{pass}$) {
    return $pass$;
  }
  else { // $s = \textbf{fail}$
    return $fail$;
  }
}

We have intentionally used the term *run* here. To test non-deterministic systems you might prefer to have multiple runs of the same test case as the test case *execution*. If any of the runs fails, then the execution fails.

In batch-mode testing the tester takes as an input a test suite, which has been created in advance. The tester executes test cases in the test suite one by one, and achieves a verdict for each test case. After execution of the test cases, the verdict for test suite can be made based on the verdicts of the individual test cases.

Automated test case generation for **ioco** is explained by the Algorithm 3.11 [9]. The algorithm non-deterministically makes a choice between all possible alternatives for a test case. In this fashion it generates one test case from the universe of all possible test cases testing **ioco** conformance.

According to the previous, the algorithm produces only test cases testing **ioco** conformance. Therefore the algorithm is be sound. As it selects test cases from the universe of all possible test cases for **ioco** conformance the set of all possible test cases generated by it has to be also exhaustive. Therefore, taking the union of all the test cases generated by all the possible runs of algorithm, a complete test suite is produced. This has been formally proved in [39].

At first glance it is not obvious that this is the case. First of all, why does the algorithm produce sound test cases for **ioco** conformance? Each state added to the set of unprocessed states SS in the algorithm has only a such trace leading to it, that the trace exists in specification. First of all this obviously holds for initial state added to SS, as there is only the empty trace, which exists

Algorithm 3.11. Let $p = (S, L_I \cup L_U \cup \{\delta\}, \Delta, s_0') \in \mathcal{LTS}(L_U \cup L_I)$ be the specification

procedure GenerateTestCase (p) {
  S := $\{s_0, \mathbf{pass}, \mathbf{fail}\}$;
  Δ := $\emptyset$;
  L := $L_I \cup L_U \cup \{\delta\}$;
  U := $\{(s_0, s_0'$ **after** $\epsilon)\}$;
  while $(U \neq \emptyset)$ {
    Take some $(s, SS) \in U$
    $U := U \setminus (s, SS)$;
    // NONDETERMINISTICALLY CHOOSE
    // BETWEEN GUARDED ALTERNATIVES
    switch {
    case of $(\exists a \in \text{init}(SS) \cap L_I)$ ->
      // ADD ARC FROM THIS STATE TO A NEW STATE WITH AN INPUT
      Select an $a \in \text{init}(SS) \cap L_I$;
      $S := S \cup \{s'\}$; // s' BEING A COMPLETELY NEW STATE
      $\Delta := \Delta \cup \{(s, a, s')\}$;
      $U := U \cup \{(s', SS$ **after** $a)\}$;
    break ;
    case of (**true**) ->
      // ADD AN ARC FOR ENABLED OUTPUTS TO NEW STATE AND
      // FOR DISABLED TO FAIL STATE
      for $\forall a \in out(SS)$ {
        $S := S \cup \{s'\}$; // s' BEING A COMPLETELY NEW STATE
        $\Delta := \Delta \cup \{(s, a, s')\}$;
        $U := U \cup \{(s', SS$ **after** $a)\}$;
      }
      for $\forall a \in (L_U \cup \{\delta\}) \setminus out(SS)$ {
        $\Delta := \Delta \cup \{(s, a, \mathbf{fail})\}$;
      }
    break ;
    case of (**true**) ->
      // REMOVE THE STATE s AND REDIRECT
      // INCOMING ARCS TO THE PASS STATE
      if $(s = s_0)$ {
        return $(\{\mathbf{pass}, \mathbf{fail}\}, L, \emptyset, \mathbf{pass})$;
      }
      $\Delta := \Delta \cup \{(s', a, \mathbf{pass}) | (s', a, s) \in \Delta, s' \in S\}$;
      $\Delta := \Delta \setminus \{(s', a, s) | (s', a, s) \in \Delta, s' \in S\}$;
      $S := S \setminus \{s\}$;
    break ;
    }
  }
  return (S, L, Δ, s_0);
}

in every specification. Secondly, cases 1 and 2 add new labels to the trace, but according to the associated specification state (denoted by SS in algorithm) only those labels are added which are enabled in specification state. And after this operation, which preserves the desired property, the state-specification state pairs are added to the set of unprocessed states U. Therefore, in U there are only states, which have only in traces existing in specification leading to them.

Our second argument is that only states existing in U are added to the test case being created (apart from pass and fail states of course). Therefore we may conclude that the traces leading to states other than pass and fail exist in specification. The case 3 transforms such state to a pass-state, and therefore traces leading to pass state exist in specification. However, transitions to fail state are added when such an output is observed, which is not enabled in a specification state associated with a state. Therefore we may conclude, that these test cases are sound.

Could there be a **ioco** nonconformant implementation i such that there is no test case generated by the Algorithm 3.11 detecting the defect? The answer should be no, if we wish that the set of all test cases generated by the algorithm is exhaustive.

Assume that there is implementation i such that it does not conform in the **ioco** sense to the specification s. Then, there must be a trace σ, such that out(i **after** σ) $\not\subseteq$ out(s **after** σ). Now however due to non-determinism Algorithm 3.11 is able to generate a test case containing state s, such that for initial state of test case s_0 it holds that $s_0 \stackrel{\sigma}{\Longrightarrow} s$, and for s it holds that for any output not in out(s) there is a transition to the fail-state. Therefore we conclude that test case detecting any fault can be generated and therefore the union of all the test cases is exhaustive.

3.2 On-the-fly Testing

Generating test cases with Algorithm 3.11 is a computationally expensive task. It requires one to consider many situations, which never actually occur during testing. Regardless of the method used, exploring the high level specification may cause a state space explosion [40] to occur. On-the-fly testing has the advantage of reducing the set of states, which have to be considered. It gets outputs from the implementation immediately and therefore may ignore all other possibilities immediately [9].

In on-the-fly testing no complete test suite is derived. Tester proceeds with the knowledge of specification and all the time selecting inputs and verifying the outputs from the specification. The tester does not have to consider any situations, which do not arise during testing. On the other hand, it has to be able to process the specification real time, as it cannot rely on previously computed information, and as previously mentioned even traversing the specification may be computationally intensive due to the state space explosion. We will present more details of on-the-fly testing, including algorithms, in the Chapter 5.

4 COVERAGE

Coverage of test events run by the on-the-fly tester is a concept to measure the quality of the testing done. It should give us a measure of the amount of testing done so that we can be confident enough in the IUT achieving the desired quality level. There has been a considerable amount of previous work with coverage, see for example [3, 6, 42]. In this chapter, we review some background and existing approaches to coverage, and then introduce our framework for coverage, partially based on these existing approaches.

In software testing community there are several coverage metrics in use. These include for example *statement coverage*, *branch coverage* and *path coverage*. Statement coverage is the percentage of executed statements under some test. Achieving 100% branch coverage means that all choices in branches have been made in some test. Path coverage measures percentage of executed control flow paths versus all possible program control flow paths [3] [3].

```
if (x > 0) then
  x++;
else
  x−−;
if (y > 0) then
  y++;
else
  y−−;
if (z > 0) then
  z++;
else
  z−−;
```

FIGURE 9: Example source code for path coverage versus branch coverage

```
if (x < 0) then
  x = x + a;
return x;
```

FIGURE 10: Example source code for statement and branch coverage

What is the difference between these coverage metrics? Path coverage considers all paths from start to the end of the program. This includes all combinations of choices, e.g., if the program contains three independent (see Fig. 9) choices, there are eight paths. Branch coverage considers paths, which choose every possible alternative on each branch. For example, if a program has three independent choices, then branch coverage may be

[3] This informal definition does not make it clear how to apply path coverage with nonterminating programs, however, this generalization is out of the scope of this work.

achieved with two executions (compare this with eight executions required for achieving path coverage). Difference between statement and branch coverage is more subtle. Consider the program in Fig. 10. Say we have a specification for this piece of code, when $x < 0$ then return $x + a$ otherwise return $x - a$. Now executing this once with $x < 0$ executes all statements once and returns the correct value, however there is a bug hidden, which would be revealed by achieving the branch coverage.

When applied to formal conformance testing, a coverage metric should tell us the level of confidence in the IUT behavior conforming to the specification. The problem is to formulate the coverage in a sensible way. Intuitively the coverage should increase as the possibility of detecting an erroneous implementation increases. Coverage level should measure the level of confidence in the conforming implementation with respect to the specification. For example a customer might require one to test 50% of the statements (w.r.t. statement coverage) to believe that there has been a reasonable amount of testing done.

In this section we denote the set of all of suspension trace sets with Ω. This is defined as follows.

Definition 4.24. *Let L be the set of labels, then the set of all suspension trace sets w.r.t. a specification s is* $\Omega = \{\Gamma \mid \Gamma \subseteq \mathit{Straces}(s)\}$.

To be more exact on what a coverage metric is, we define it as a function from the set of all suspension trace sets to real numbers from the closed interval $[0, 1]$. This idea fits to the concept of black box testing, as the traces are the only observations that we make on the implementation. Unfortunately, for general white box testing we need to take int oaccount the internal choices of the implementation and thus this definition does not apply in general for white box testing. In black box testing it applies, but we have to actually employ a family of functions. For example, we might have a different coverage metric function for each specification. From now on we will loosely use the term “coverage metric” to denote also such a family of coverage metrics.

A coverage metric function must be defined carefully, so that for all $\Gamma \in \Omega$ there exists an image in $[0, 1]$. We denote closed interval between 0 and 1 with $[0, 1]$, formally $[0, 1] = \{r \in \mathbb{R} \mid 0 \leq r \leq 1\}$, where $\mathbb{R}$ is the set of real numbers.

Definition 4.25. *Coverage metric C is a function* $C : \Omega \rightarrow [0, 1]$*, where* Ω *is the set of all suspension trace sets as defined in Def. 4.24.*

A set of traces achieves the coverage metric iff we reach a maxima of the coverage metric in question with the set of traces. For **ioco** conformance we consider only the suspension traces of the specification.

Definition 4.26. *Let s be a specification. Coverage metric C is achieved for this specification by the set of suspension traces* $\Gamma \in \Omega$*, iff* $\forall \Gamma' \in \Omega : C(\Gamma) \geq C(\Gamma')$.

A necessary condition we impose on a coverage metric to be well defined is the existence of such maxima. Optimal trace set for a coverage metric is a set of traces with the smallest total length achieving the coverage metric. Next we define the finite optimal trace set, as the definition for infinite optimal trace sets is cumbersome. Note that a finite optimal trace set does not always exist neither is it unique.

Definition 4.27. $\Gamma_C^O \subseteq \Omega$ *is a finite optimal trace set for coverage metric* C *w.r.t. to a specification s iff: (i)* Γ_C^O *is finite, (ii)* $\forall$ *finite* $\Gamma \in \Omega : \sum_{\sigma \in \Gamma_C^O} |\sigma| \leq \sum_{\sigma \in \Gamma} |\sigma|$*, and (iii) coverage metric* C *is achieved by* Γ_C^O*.*

Intuitively a coverage metric should be monotonic, as executing more tests should not decrease the confidence in conforming implementation if they do not fail. We define this property for a coverage metric as follows.

Definition 4.28. *Coverage metric* C *is monotonic if and only if* $\forall\, \Gamma \in \Omega : \forall\, \Gamma' \subseteq \Gamma : C(\Gamma') \leq C(\Gamma)$.

By definition a complete test suite is able to detect a defective implementation. By definition one might be misled to think that executing such test suite in a way that it detects all the faults would be easy. We are however considering black box testing, which in practice hides these details from the tester. Although the implementation may be modeled as LTS, we do not know any explicit states, transitions or labels. We just observe events. Such an implementation might also be non-deterministic. If this is the case, one may not ever know, whether there is still some non-conforming behavior which has not been observed, although all possible tests have been executed. Such problems occur only when we do not know the internals of the implementation and the actual states and transition relation are therefore unknown. If we know the internals, we can compute the all possible traces without even testing the implementation. Or, if we do not wish to compute all the traces we may compute the resulting state after some particular trace and compute the out set of this state. However, in some cases the computation might need excessive resources.

So we conclude that, applying a theoretically complete test suite has two major problems to be considered. First of all, when testing a non-deterministic black box implementation, one may not be able to obtain all the traces of such an implementation. Therefore by executing a complete test suite made on the basis of the theory presented above may not detect all defects, as one might wrongly assume on the basis of the definition of complete test suite. Secondly, such a test suite is in general infinite and therefore impossible to generate or to execute with finite testing resources, which in practice is always the case. Therefore executing or generating such a test suite would not be finished in finite time with finite computing capacity.

However, in certain aspects this the best we can do. After all, infiniteness of test suite is direct consequence of the **ioco** -definition. Detecting faults of a non-deterministic system, which will not exhibit non-conforming behavior when testing it, is also obviously impossible. Therefore we will, for now, assume that best we can do is to select an appropriate subset of complete test suite and assume that non-deterministic implementations will not prevent us from detecting bugs. From now on, we will by default forget the problems related to this non-determinism.

In this work we model the observations during testing with the trace set, for which we use the symbol Γ. Precise definition how this set is constructed during testing is given in Chapter 5 with the algorithms for testing, but in general on-the-fly tester tests an implementation by sending inputs to implementation and receiving the outputs. Sequences of these inputs and outputs are suspension traces, when the suspension is considered as output. The on-the-fly tester may also reset the implementation. When a reset happens, the current test sequence is terminated and new sequence is started. This is how the traces are formed. Alternatively we could have used traces with reset symbols. Such symbols have been used in [21].

A coverage metric corresponding to executing a complete test suite would be achieved by a trace set resulting from executing a complete test suite, for example one produced by the Algorithm 3.11. Def. 2.14 involves the out sets after all traces. First we assume the following: If the implementation may exhibit non-conforming behavior after some trace, then it will do so every time. This assumption is questionable for non-deterministic implementations, but for now we do not concentrate on that issue. Such implementations occur in practice, but we do not focus on them now for the sake of simplicity[4].

As a complete test suite is able to detect all the bugs, we consider it a good idea to have a coverage metric which resembles a complete test suite in the sense that achieving it corresponds to executing the complete test suite in question. Based on the previous discussion we can conclude that the set of all suspension traces achieves the coverage metric corresponding to the complete test suite.

In general the set of suspension traces of any non trivial specification is infinite. We define *Trace coverage* of a trace set as a sequence which converges towards 1, when the trace set approaches $Straces(\mathrm{s})$. It is defined in a way that the longer a trace is, the less it affects the coverage. However, the trace coverage is not achieved until all traces have been observed. The sequence contains terms, which represent the proportion of traces of length k whose continuation (continuation is a trace whose prefix a trace is) has been observed. These terms are scaled with factor $\frac{1}{\alpha^k}$, where k is the length of trace, to make the series converging.

[4]Without this assumption it might be reasonable, for example, to define coverage in such manner, that it would increase when a trace would be executed several times to gain more confidence on the conforming behavior of nondeterministic implementations. This could be implemented using a multiset based coverage, i.e., coverage where each trace should be executed n times to reach the full coverage.

Definition 4.29. *Let s be a specification and Γ a trace set. Let $Pref(\Gamma)$ be the prefix closure of this set, defined as $Pref(\Gamma) = \{\sigma \mid \exists \sigma' \in (L \cup \{\delta\})^* : \sigma \cdot \sigma' \in \Gamma\}$. Based on this we define as follows*

$$\begin{aligned} N_k &=_{def} \quad |\{\sigma \in Straces(s) \text{ such that length of } \sigma \text{ is } k\}| \\ C_k &=_{def} \quad |\{\sigma \in Pref(\Gamma) \text{ such that length of } \sigma \text{ is } k\}| \end{aligned}$$

and based on these

$$\pi_k =_{def} \begin{cases} 1 & \text{if } N_k = 0 \\ \frac{C_k}{N_k} & otherwise. \end{cases}$$

The trace coverage of set Γ w.r.t. s is

$$C_T(\Gamma) \quad =_{def} \quad (\alpha - 1) \sum_{k=1}^{\infty} \frac{1}{\alpha^k} \pi_k.$$

The series converges if the geometric series $\sum_{k=1}^{\infty} \frac{1}{\alpha^k}$ converges. This is the case when $\alpha > 1$. Observing any unobserved prefix increases the coverage, as it increases proportion of observed traces of this length. For example, if we have not observed any continuation of σ with length n, then the coverage is $(\alpha - 1)\frac{1}{\alpha^n N_k}$ lower than if σ had been observed.

This definition has advantages that first of all it is monotonic and secondly it has a maxima with value 1, when $\Gamma = Straces(s)$. It also works for infinite traces and is quite simple. However, note that this definition should not be interpreted as percentage although it has values between 0 and 1.

The coverage contains parameter α to make it applicable to variety of cases where e.g. we may assume that the shorter traces are more important than the longer. One may select α according how much the shorter traces are more important than the longer. As the α gets larger the short traces become more important. If α gets smaller the long traces get more weight in the calculation.

Example 4.12. Consider our running example from Section 2 in Fig. 1. This example has following traces with length smaller than 5: $\{a, ac, ab, a\delta, aca, aba, a\delta\delta, acab, acac, aca\delta, abac, abab, aba\delta, a\delta\delta\delta\}$. This results 1 trace of length 1, 3 of length 2 and 3 of length 3 and 7 of length 4. Assume we have executed the following traces: $\{a, ab, aba, abab\}$. If we set $\alpha = 2$ then the resulting trace coverage of this set is about 0.63. Compare this to the case where we set $\alpha = 4$, the trace coverage is about 0.83. For trace set $\{a, ab, ac, a\delta\}$ the trace coverages are 0.75 ($\alpha = 2$) and about 0.94 ($\alpha = 4$).

We do not have a practical motivation for the factors α except that obviously conforming behavior on smaller traces is most important. However, the target is to achieve trace coverage. It would in general mean executing an infinite amount of test cases. Therefore it is not a practical option. However, we may at least approximate it in the following sense: we define an approximating coverage metric in such a way that increasing it guarantees that the trace coverage metric increases.

Another possibility to handle the problems related to infiniteness of the complete test suite and non-determinism is to define a *fault model*. A fault model specifies which kind of defective implementations we expect to have. We might define the following fault model for example: We target only detecting implementations producing fault with traces up to length n and having no non-determinism. This way, we are able to assure ourselves that we have no faults w.r.t. the fault model. Defining reasonable fault models is not straightforward. Examples of fault models for hardware, software and protocol testing are presented in [6].

For fault models one may define a probabilistic coverage metric with a formula, which calculates the conditional probability of verdict fail with the condition that implementation is faulty. This is called effective coverage of test suite, and for a test suite T we define it as $\mathrm{EC}_T = \frac{\sum_{i \in I_F} P_{Fail}(T,i) P_i}{\sum_{i \in I_F} P_i}$, where I_F is the set of faulty implementations, P_i is the probability of implementation being i, and $P_{Fail}(T, i)$ is the probability that test suite T fails when applied to the implementation i [6]. If we also assume that the test suite is sound, this would be a very good coverage metric. Unsound test suites with 100% effective coverage are easy to generate as they contain test cases always producing the fail verdict.

The hard part here is to define the probabilities for failing test cases, and implementation being some particular implementation. Restricting ourselves to some type of faults, and statistical analysis of typical faults in application domain might help, but still makes this approach harder than it first looks.

Also interesting approach can be found in [2]. They generate tests by generating mutants from the implementation with mutation operators and then try to find counterexamples proving the mutations non conforming with model checkers SPIN[22] and SMV[27]. Operators generating mutants can be thought as fault models.

They further even refine this in [1] to include specification based coverage. However, this specification-based coverage is pretty different from ours. They define it as the amount of mutants detected by test case set per all mutants produced by mutation operators.

We will base our work on trying to increase trace coverage metric using simple specification-based coverage metrics and moreover it is also inspired by the branch coverage, and therefore our approach is also similar to those used already in software testing community. As an interesting side note, we can also see similarities between the trace coverage metric and path coverage, and note that increasing branch coverage increases path coverage. In next section we will define our coverage framework for on-the-fly testing in detail.

4.1 Coverage Framework for On-the-fly Testing

We define a coverage metric on three different levels: The suspension LTS level, the labelled transition system level and on the labelled Petri net level. For example, we may define coverage metric based on the state coverage or the transition coverage. However, in this work we concentrate only on transition coverage.

Let $p = (S, L, \Delta, s_0)$ be a LTS specifying some system. We can see the traces of this system from two different perspectives: a practical and a theoretical perspective. The traces from a theoretical perspective are $Straces(s_0)$, and from a practical perspective they are the sequences of events between the IUT and the tester. A trace from practical viewpoint starts either by beginning the testing or resetting the implementation. After a trace starts all the events are included in it in sequence up to the reset or the ending of the testing. All the traces observed during testing form the trace set Γ.

All the three transition coverage metrics to be defined are similar to the branch coverage. Different levels can be intuitively thought as different levels of abstraction. The smaller the specification is the higher the level of abstraction is. Less abstract levels approximate trace coverage more closely.

In the transition coverage of a Petri net and transition coverage of a LTS we do not try to handle the suspensions. The reason for this is simplicity. However in suspension automaton coverage we cover the suspensions. In following definitions we assume that the specification Petri net and specification LTS have at least one visible transition, i.e., $|\Delta_v| \geq 1$ and $|vistrans(N)| \geq 1$. All useful specifications fulfill this criteria.

Definition 4.30. *Let $N = (P, T, F, \lambda, M_0)$ be a specification Petri net, and let $LTS(N) = (S, L, \Delta, s_0)$ be the induced LTS. A suspension trace $\sigma \in (L \cup \{\delta\})^*$ covers the visible transition $t \in vistrans(N)$ iff there exists $\sigma', \sigma'' \in (L \cup \{\delta\})^*$, $s \in S$, and $M, M' \in RM(N)$ such that:*

(i) $\sigma = \sigma' \cdot \lambda(t) \cdot \sigma''$,

(ii) $s_0 \overset{\sigma'}{\Longrightarrow} s = M$, *and*

(iii) $M \xrightarrow{t} M'$.

The set of covered transitions by a suspension trace σ is denoted as covered(N, σ), and by a suspension trace set Γ as covered$(N, \Gamma) =_{def} \bigcup_{\sigma \in \Gamma}$ covered(N, σ).

Definition 4.31. *Transition coverage of a labelled Petri net N w.r.t. trace set Γ is defined to be $C_T(N, \Gamma) = \frac{|\text{covered}(N,\Gamma)|}{|vistrans(N)|}$.*

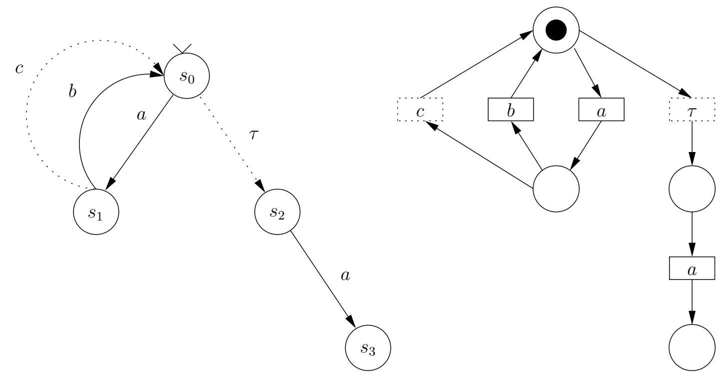

FIGURE 11: Petri net transition coverage of trace ab as a LTS (left) and a Petri net (right)

Example 4.13. The Petri net transition coverage of trace ab w.r.t. our running example is presented in Figure 11. The running example is a LTS, but there is trivial transformation producing corresponding labelled 1-safe Petri net, which is formed from LTS by considering states as places and arcs as transitions. This is shown on the right. Now the arcs with solid edges are covered and dotted arcs are not. In the Petri net covered transitions have solid edges around them and uncovered ones are dotted. Note that internal transitions will never be covered.

Definition 4.32. *Let σ be a suspension trace and $p = (S, L, \Delta, s_0)$ be the specifying LTS. A visible arc $(s, a, s') \in \Delta$ is covered by the trace σ iff there exists $\sigma', \sigma'' \in (L \cup \{\delta\})^*$ such that*

(i) $\sigma = \sigma' \cdot a \cdot \sigma''$,

(ii) $s_0 \stackrel{\sigma'}{\Longrightarrow} s$, *and*

(iii) $s \stackrel{a}{\longrightarrow} s'$.

The set of covered arcs in LTS p by a trace σ is denoted as covered(p, σ) and by a trace set Γ as covered$(p, \Gamma) =_{def} \bigcup_{\sigma\in\Gamma}$ covered(p, σ).

Definition 4.33. *LTS $p = (S, L, \Delta, s_0)$ visible transition coverage w.r.t. trace set Γ is defined to be $C_T(p, \Gamma) = \frac{|\text{covered}(p,\Gamma)|}{|\Delta_v|}$.*

For a suspension automaton the definitions are almost the same as for an LTS, except that the original LTS is replaced by the suspension automaton. Therefore this coverage is in general different from the original LTS coverage.

Definition 4.34. *Let σ be a suspension trace and SA$(N) = (S, L, \Delta, s_0)$ the suspension automaton of the specification. A visible arc $(s, a, s') \in \Delta$ is covered by the trace σ iff there exists $\sigma', \sigma'' \in (L \cup \{\delta\})^*$ such that*

(i) $\sigma = \sigma' \cdot a \cdot \sigma''$,

(ii) $s_0 \stackrel{\sigma'}{\Longrightarrow} s$, *and*

(iii) $s \stackrel{a}{\longrightarrow} s'$.

The set of covered arcs in SA(N) by a trace σ is denoted as covered(p, σ) and by a trace set Γ as covered$(\text{SA}(N), \Gamma) =_{def} \bigcup_{\sigma\in\Gamma}$ covered$(\text{SA}(N), \sigma)$.

Definition 4.35. *The suspension automaton (of specification N) visible transition coverage w.r.t. trace set Γ is defined to be $C_T(\text{SA}(N), \Gamma) = \frac{|\text{covered}(SA(N),\Gamma)|}{|\Delta_v|}$, where $|\Delta_v|$ refers to the number of visible arcs in the suspension automaton. Note that the δ is a visible symbol of the suspension automaton.*

We remind that we have already defined trace coverage of a trace set in the Def. 4.29.

Proposition 4.1. All previous coverage metrics (including trace coverage) grow monotonically, as the trace set Γ grows.

Theorem 4.2. Consider a trace sets Γ and Γ' such that $\Gamma \subset \Gamma'$. Now if $C_T(N, \Gamma') > C_T(N, \Gamma)$ holds for a labelled Petri net N, then $C_T(\text{LTS}(N), \Gamma') > C_T(\text{LTS}(N), \Gamma)$.

Proof This is a proof by contradiction. Let Γ' be some trace set and $\Gamma \subset \Gamma'$. Assume following holds for a labelled Petri net $N = (P, T, F, \lambda, M_0)$:
$C_T(N, \Gamma') > C_T(N, \Gamma)$ and $C_T(\mathrm{LTS}(N), \Gamma') \leq C_T(\mathrm{LTS}(N), \Gamma)$. By monotonicity it does not hold that $C_T(\mathrm{LTS}(N), \Gamma') < C_T(\mathrm{LTS}(N), \Gamma)$ and thus $C_T(\mathrm{LTS}(N), \Gamma') = C_T(\mathrm{LTS}(N), \Gamma)$. Because the total number of visible transitions remains the same in both cases, $\exists t \in T$, covered by some $\sigma \in \Gamma' \setminus \Gamma$, but not covered by any trace in Γ. However, by our assumption σ does not increase the LTS transition coverage. It follows that for each arc (s, a, s') covered by σ there is a trace $\sigma' \in \Gamma$ which covers it. However, this means that each Petri net transition covered by σ is covered with some trace $\sigma' \in \Gamma$ and therefore we have a contradiction and the theorem follows. This is due that for each arc there exists one transition in the Petri net which is covered by all the traces covering the arc. $\square$

Theorem 4.3. Consider trace sets Γ and Γ' such that $\Gamma \subset \Gamma'$. Now for the Petri net N induced LTS $\mathrm{LTS}(N)$ following holds: if $C_T(\mathrm{LTS}(N), \Gamma') > C_T(\mathrm{LTS}(N), \Gamma)$ then $C_T(\mathrm{SA}(N), \Gamma') > C_T(\mathrm{SA}(N), \Gamma)$.

Proof Some trace $\sigma \in \Gamma$ covers an arc (s, a, s') in an LTS p, which is not covered by Γ. Obviously a prefix σ' of σ such that $s \in S = p$ **after** σ'. In the suspension automaton we have a state S such that $s_0 \stackrel{\sigma'}{\Longrightarrow} S$ and an arc (S, a, S') (where $S' = S$ **after** a) which has not been covered by Γ. This due the fact that, if Γ had covered this, then it would have covered also the arc (s, a, s') in the LTS. $\square$

Theorem 4.4. Consider trace sets Γ and Γ' such that $\Gamma \subset \Gamma'$. For the suspension automaton $\mathrm{SA}(N)$ following holds: If $C_T(\mathrm{SA}(N), \Gamma') > C_T(\mathrm{SA}(N), \Gamma)$, then $C_T(\Gamma') > C_T(\Gamma)$.

Proof First of all, there must exist a trace σ in the set Γ' such that it covers arcs in the suspension automaton which are not covered by any trace belonging to Γ. Assume that all the prefixes of σ are also prefixes of some trace in the set Γ. However this would mean that all the arcs of the suspension automaton covered by σ would be covered by Γ too. Therefore this is a contradiction and σ has to have a prefix which is not a prefix of any trace in Γ.
Secondly, all the prefixes of traces in Γ are included in prefixes of Γ'. This is obvious as $\Gamma \subset \Gamma'$. Therefore we conclude that the Γ' contains more unique prefixes that Γ and the trace coverage is thus higher. $\square$

Based on Theorem 4.2, Theorem 4.3, Theorem 4.4 we may conclude that increasing any of the following three coverage metrics (i) transition coverage of a labelled Petri net, (ii) LTS transition coverage, or (iii) suspension automaton transition coverage, increases the trace coverage. Increased trace coverage increases our chances to detect bugs.

5 ALGORITHMS

In this section we study conformance testing algorithms for on-the-fly testing. We present a general framework into which we fit our approach and existing on-the-fly testing algorithm by Tretmans et al [9]. The extension we present allows us to build heuristics for this algorithm. After algorithms we consider what properties could be used for analyzing the capabilities of these algorithms. We adapt the definitions of soundness, exhaustiveness and completeness for on-the-fly testing algorithms and define the omnipotentness. Then we consider which properties the algorithms presented satisfy.

5.1 A Conformance Testing Algorithm

First we present Algorithm 5.14, which is our general on-the-fly testing algorithm. This algorithm lets the heuristics to be defined later. The algorithm returns fail when it has detected that the implementation has failed to conform to the specification and pass otherwise.

We use here a few data objects worth of describing. First of all a *super state* is a collection of states. Often specification contains some nondeterminism, i.e., some sequence of actions does not lead to some specific state, but possibly to many different states. As we model all the behavior contained in the specification we take account all of these states and use super states to analyze the specification behavior.

We also use data objects called traces and trace sets. Trace sets are set of traces, but what is trace? It is a suspension trace from Chapter 2, i.e., a sequence of events describing the observed behavior of the implementation. By ϵ we denote an empty trace containing no events and by $\emptyset$ we denote empty trace set, containing no traces.

By defining a random heuristic as in Algorithm 5.15 we get the on-the-fly testing algorithm of [9] presented from an implementation perspective. The resulting algorithm is actually very similar to Algorithm 3.11 which is able to generate a test suite for **ioco** conformance testing based on specification as LTS. The algorithm can be considered as the on-the-fly testing version of the Algorithm 3.11.

In Algorithm 5.14 we model the general on-the-fly test process. With *TestMove* algorithm we present the choice of next action. The actual heuristic lies behind this routine. By making a random choice as in Algorithm 5.15 we model non-deterministic choice. We also present more sophisticated heuristic in Algorithm 5.16.

Apart from deciding the next action with guidance from heuristics, the algorithm also sends inputs to implementation and observes outputs from it. We use similar notation to Chapter 3. By *Stimulate*(i,x) we denote that implementation i is sent an input x and the subroutine *ObserveOutput*(i) returns the output observed from the implementation i including suspension δ. Suspension δ is caused by situation where no output was observed.

The algorithm also keeps track of tests executed. In this task we use *TestLog*() subroutine which is introduced in detail in Algorithm 5.19. It basically adds executed events to currenttrace, and completed traces to executedtraces.

Algorithm 5.15 represents a heuristics which is implemented to give a random answer. It has constants $Prob\_input$, $Prob\_reset$ and $Prob\_terminate$ to denote the probabilities of giving input, resetting the implementation and terminating. These constants may affect the performance of testing, for example see [15]. The subroutine *TrueWithProbability* returns true with probability given as parameter and false with the complement probability.

Algorithm 5.14.

```
procedure OnTheFlyTest(IOTS(L_I ∪ L_U) i, LTS(L_I ∪ L_U) s) {
  currenttrace := ε;
  executedtraces := ∅;
  S := {s_0}; // AT BEGINNING CURRENT SUPER
  // STATE CONTAINS THE INITIAL STATE s_0 OF THE SPECIFICATION S
  S := S after ε;

  while (true ) {
    move := TestMove(currenttrace, executedtraces, S); // RETURNS EITHER
    // MOVE ∈ L_I OR MOVE ∈ {output, terminate, reset}
    switch (move) {
    case of terminate:
      TestLog(currentttrace, executedtraces, S, pass);
      return pass;
    case of reset:
      TestLog(currenttrace, executedtraces, S, reset);
      break ;
    case of move ∈ L_I:
      // TestMove() INSTRUCTED TO MAKE AN INPUT
      x := move; // TestMove() MUST GUARANTEE THAT x ∈ (init(S) ∩ L_I)
      Stimulate(i, x); // STIMULATE THE IMPLEMENTATION WITH INPUT X
      TestLog(currenttrace, executedtraces, S, x); // LOG THE INPUT SENT
      S := S after x;
      break ;
    case of output: // MOVE = output
      // TestMove(S) INSTRUCTED TO OBSERVE AN OUTPUT
      x := ObserveOutput(i);

      if (x ∈ out(S)) then { // HANDLES SUSPENSIONS TOO
        S := S after x;
        // LOG THE OUTPUT OR TIMEOUT OBSERVED
        TestLog(currenttrace, executedtraces, S, x);
      } else {
        // AN OUTPUT WAS OBSERVED, BUT THE SPEC CAN NOT MAKE IT
        TestLog(currenttrace, executedtraces, S, fail);
        return fail;
      }
      break ;
    }
  }
}
```

FIGURE 12: The on-the-fly testing algorithm main loop

Algorithm 5.15.

```
procedure RandomTestMove(SuperState S) {
  inputs := init(S) ∩ L_I;
  if (TrueWithProbability(Prob_terminate) then {
    return terminate;
  }
  elsif (TrueWithProbability(Prob_reset) then {
    return reset;
  }
  elsif (inputs ≠ ∅ ∧ TrueWithProbability(Prob_input)) then
    move := PickRandomElement(inputs);
  } else {
    move := output;
  }
  return move;
}
```

FIGURE 13: Implementation of test move making random moves

The idea in our improved *TestMove*(Algorithm 5.16) is to pick a random action by small probability or with a higher probability to greedily choose an action which is likely to lead to a higher coverage. Why we have the small chance to choose randomly a test move is to make our testing algorithm complete, i.e., being able to detect any bug in the implementation. It is also possible to tune parameters to have greater chance for randomly selecting the action, however, we see this kind of parameter setting less likely.

Greedily choosing an action is done first by looking for an immediately enabled transitions, and if there is no such transition leading to a higher coverage, using the *Bounded Model Checking* (BMC) [5] technique to look ahead more than one step. BMC returns a move of sequence leading to this uncovered transition. However, the sequence is not guaranteed to lead to increasing coverage as it may contain some output transitions. The output transitions are not guaranteed to fire, as they are controlled by the implementation, i.e, it may send some other output instead of the one we were wishing for. How the BMC exactly computes the sequence will be explained later in 6.3.

The heuristic may also reset the implementation or terminate testing. The heuristic terminates testing when either reaching full coverage or executing the amount of testing events specified by user as the bound. User may also set a reset interval, i.e., implementation is reset after n events, where n is a bound chosen by the user of our tool.

The look ahead is straightforward. First the look ahead queries BMC for an execution leading to higher coverage. The execution is restricted by an user specified bound on the length and therefore such an execution is not guaranteed to exist. If such execution is found, heuristic stores this execution and returns an action to do the first step of the execution. It is either sending an input or waiting for an output. The latter lets the implementation to disturb our newly found execution, as it may return any output, not the one

Algorithm 5.16.

```
procedure HeuristicTestMove(
Trace currenttrace, TraceSet executedtraces, SuperState S) {
  move := none;
  if ( |executedtraces| > maxruns ∨ coverage > requiredcoverage) {
    return terminate;
  }
  if (|currenttrace| > resetInterval) {
    return reset;
  }
  if (TrueWithProbability(Prob_greedy)) then {
    move := GreedyTestMove(S);
  }

  if (move = none) then {
    move := RandomTestMove(S);
  }

  return move;
}
```

FIGURE 14: Heuristic test selection combining greedy and random elements

we had in our execution. If an execution is not found it returns *none* which causes the *RandomTestMove* to be called in the callee function.

During testing the algorithm Algorithm 5.14 keeps track of the traces executed. The set executedtraces equals to the set of executed traces Γ in previous section and therefore this is how that set is exactly obtained. Every executed event is appended to current trace and once the implementation i s reset current trace is added to the set of executed traces. This happens in TestLog() subroutine (Algorithm 5.19), which also updates the coverage information.

Updating the coverage information means marking new covered transitions as covered. The covered transitions by some trace are specified in Def. 4.30. In this algorithm we only have the current super state and the new event. However, this information is sufficient to update the coverage as the previously covered transitions have already been marked covered. By combining the information of the current super state and the event executed we know which transitions are covered by this event. If S is the current super state, then the event x covers all visible transitions on every path $s \overset{x}{\Longrightarrow} s'$, where $s \in S$ and $s' \in (S \textbf{ after } x)$.

5.2 Soundness and Completeness of Heuristics

When classifying algorithms, we must first define the notion of the algorithm. Algorithm is understood to be something, which can be computed (for example by a Turing machine) with a finite number of steps. However, we use this term in this section loosely and do not require our algorithms to

Algorithm 5.17.

```
procedure GreedyTestMove(SuperState S) {
  input_uncovered := UncoveredInputIsEnabled(S);
  output_uncovered := UncoveredOutputIsEnabled(S);

  if (input_uncovered ∧ output_uncovered) then {
    if (TrueWithProbability(Prob_input)) then }
      choice := input;
    } else {
      choice := output;
    }
  } elsif (input_uncovered ∧ ¬output_uncovered) then {
    choice := input;
  } elsif (¬input_uncovered ∧ output_uncovered) then {
    choice := output;
  } else {
    choice := none;
  }

  if (choice = input) then {
    move := PickRandomUncoveredInput(S);
  } elsif (choice = output) then {
    move := output;
  } else {
    move := LookaheadTestMove(S);
  }

  return move;
}
```

FIGURE 15: Coverage based greedy test selection subroutine

terminate in a finite number of steps. Such algorithms are often called semi-algorithms. The semi-algorithms might not terminate with some inputs, but algorithms (in strict sense) do terminate with every input after a finite number of elementary operations (for example transitions in an execution of a Turing machine). Note that every algorithm is also a semi-algorithm. In this section if we say that algorithm returns a value, it also means that it has terminated.

We use notion $a(x, y, z) = value$ to denote that algorithm a with inputs x, y and z terminated returning value $value$. In the case that algorithm does not terminate, we define that it returns a special value $notterminated$.

In this subsection we will restrict ourselves to a deterministic implementation, by which we mean that it's behavior is determined by the inputs sent only (no internal non-determinism). This simplifies the analysis, because non-deterministic implementations complicate most of the aspects.

In this subsection we wish to define some concepts to classify on-the-fly testing algorithms. We define four classes of algorithms: sound, exhaustive, complete and omnipotent. First three are as previously defined for the test suites, and the last is to denote an algorithm which might detect any bug, but

Algorithm 5.18.

procedure *LookaheadTestMove*(SuperState S) {

With limited computational resources try to find a (preferably) short execution: $S \xRightarrow{\sigma} S'$, such that any test run beginning with σ increases coverage.

move := $none$;
if (σ was found) **then** {
Let x be the first action of σ
if ($x \in L_I$) **then** {
move := x;
} **else** {
move := $output$;
}
}

return move;
}

FIGURE 16: An abstract subroutine description for coverage based lookahead

the conclusion of implementation being conforming does not guarantee that the implementation is conforming.

The presented algorithms depend on random numbers. Random numbers in algorithms are interesting topic of their own, and interested reader is pointed to for example [28, 26]. Basically we have pseudo-random and true-random bits and also something between. The random number generators available through "random()" library routines in commonly used programming languages are based on on mathematical algorithm producing "random" numbers from a finite initial seed. The sequence is a function of the seed, and therefore not random. However, in this theoretical treatment of testing algorithms we consider a random bit source such that the bits are truly random. Such source is random variable whose realizations are infinite sequences of bits $\{x_1, x_2, \ldots\}$ where $\forall i > 0 : P(x_i = 0) = 0.5$ and $\forall i > 0 : P(x_i = 1) = 0.5$. One must not be able to conclude anything by the previous bits either, and therefore $\forall i > 0 : \forall Y = y_1 \cdot y_2 \cdots y_{i-1} : P(x_i = 0 | x_1 \cdot x_2 \cdots x_{i-1} = Y) = 0.5$ as well as $\forall i > 0 : \forall Y = y_1 \cdot y_2 \cdots y_{i-1} : P(x_i = 1 | x_1 \cdot x_2 \cdots x_{i-1} = Y) = 0.5$.

To analyse algorithms including random variables we introduce two views, to view them either as deterministic or non-deterministic algorithms. The essence is to model our algorithms, so we can model same algorithm in different ways. Deterministic algorithm is deterministic when run with the same realization of the random source and same responses by the implementation to the stimuli sent by the testing algorithm. When these prerequisites are true the deterministic algorithm has a unique run.

The non-deterministic view analyses all possible runs of the algorithm corresponding to the different realizations of the random source. This view is

Algorithm 5.19.

procedure TestLog(Trace currenttrace, TraceSet executedtraces,
SuperState S, action x) {
// CURRENTTRACE AND EXECUTEDTRACES ARE PASSED BY REFERENCE
  if ($x \notin \{reset, pass, fail\}$) {
    currenttrace = currenttrace · x;
    // INITIALLY ALL TRANSITIONS ARE UNCOVERED
    Mark all transitions with label x and enabled in S as
    covered according Def. 4.30
  }
  else {
    executedtraces = {currenttrace} ∪ executedtraces;
    currenttrace = ϵ;
  }

FIGURE 17: This algorithm constructs executed traces from individual events

similar to the non-deterministic Turing machines [29]. The result of such an algorithm is pass, iff all possible realizations of the random source either (1) terminate with the verdict pass or (2) do not terminate. The result of non-deterministic algorithm is fail when the result is not pass.

Definition 5.36. *Deterministic testing algorithm a is sound if, for all implementations i and specifications s and realizations of the random source r it holds that if $a(i, s, r) = fail$, then i* $\mathbf{io\!\!\!/co}$ *s.*

Definition 5.37. *Deterministic testing algorithm a is exhaustive if, for all implementations i and specifications s and realizations of the random source r it holds that if $a(i, s, r) = pass$ then i* **ioco** *s.*

Definition 5.38. *Deterministic testing algorithm a is complete if it is sound and exhaustive.*

Definition 5.39. *Deterministic testing algorithm a is omnipotent, if it is sound and when i* $\mathbf{io\!\!\!/co}$ *s then there is a realization of the random source r such that $a(i, s, r) = fail$.*

Example 5.20. Trivial sound algorithm is "return pass" and trivial exhaustive algorithm is "return fail". Obviously you cannot get trivial complete algorithm by combining these. Actually no algorithm can guarantee termination and completeness at the same time. Trivially one can just not terminate and that is a complete testing algorithm, although not very useful. More interesting is a complete algorithm (possibly non-terminating), that systematically asks for an output after every possible suspension trace and therefore does not terminate for specifications with an infinite number of suspension traces. Actually, all specifications happen to have an infinite number of suspension traces, however, by limiting ourselves to suspension traces having only one suspension in a row we get the things intuitively right. Omnipotent algorithm is produced by adding the possibility terminate with some probability with verdict "pass" to the non-trivial complete algorithm.

The Algorithm 5.14 produces fail, only when after some trace σ we either observe an output $x \notin \text{out}(s\ \ \textbf{after}\ \sigma)$ or absence of output $\delta \notin \text{out}(s\ \ \textbf{after}\ \sigma)$ therefore we may conclude that w.r.t. **ioco** conformance relation our algorithm is sound. This result is independent of heuristics (*TestMove*() - subroutine) used.

The omnipotentness of Algorithm 5.14 depends on the heuristics used. A sufficient condition for omnipotentness is the following: there is nonzero probability for sending any enabled input or receiving an output every time, i.e., *TestMove*() subroutine has always non-zero probability to return $output$ or any of the enabled inputs.

Reset of the algorithm is not visible in the algorithm is not considered as an input or an output, but rather an "out of band" event.

Example 5.21. The omnipotentness of Algorithm 5.14 combined with Algorithm 5.16 could be achieved by setting the following parameters requiredcoverage $= 1.0$, maxruns $= \infty$ and resetInterval $= \infty$. By setting these values the algorithm has no guarantee of termination. If any of these parameters is set to a lower value, then at certain point the probability of sending any output or input becomes zero, as algorithm certainly terminates at the point of reaching the decreased value.

The exhaustiveness of Algorithm 5.14 depends on the heuristics and implementation. Recall, that we have already made restriction to the deterministic implementations. If the heuristics is omnipotent, and does not terminate until (i) all the sequences of inputs and output requests allowed by the specification have been used against the implementation (sequences start from the starting state after reset) and (ii) corresponding outputs observed: it will be exhaustive. Practically this produces algorithm, which does not terminate for a conforming IUT. If we extend our framework with the restriction that suspension traces do no contain two suspensions in a row there would be a small subset of specifications (those having a finite amount of traces) for which a complete algorithm might terminate (in a combination with conforming IUT).

Example 5.22. The previous omnipotentness example is actually also exhaustive as it never terminates when i **ioco** s! We can conclude that this is complete algorithm, as it was already known to be sound.

Practically implementing an exhaustive on-the-fly testing algorithm is awkward, because it must not terminate (in general) after a finite number of steps. As you have seen in our example, the Algorithm 5.14 does not terminate with the parameters making it complete.

We can also conclude that our heuristic gives a sound deterministic algorithm[5], which is not omnipotent, exhaustive or complete until the parameters are chosen very carefully (i.e., as in Example 5.21).

We can also make exactly same definitions for non-deterministic algorithms. If the suspension traces are executed in such manner that before any finite trace only a finite number of events are executed, then for non-deterministic algorithms the complete and omnipotential are equivalent.

[5]Possibly non-terminating algorithm.

For nondeterministic implementations we must have some assumptions, say a fairness assumption that if some input is sent infinitely many times, then all possible output behaviors can be observed. The analysis made in this section was based on assumption that implementation is deterministic.

6 IMPLEMENTATION

We have implemented an extensible testing tool called *Bomotest*. This tool is an on-the-fly testing tool, and it supports labelled 1-safe Petri nets as the input formalism. There are also several heuristics, and a coverage metric implemented. Additional related implemented utilities include a *simulated implementation under testing*, from now on *SimIUT* and *Reachability analyzer* for 1-safe Petri nets. SimIUT is a tool for simulating an implementation specified with 1-safe Petri net. The 1-safe Petri net, which is given as a parameter to SimIUT, specifies the behavior of the implementation to be simulated. It can be attached to Bomotest and it responds to stimuli according the 1-safe Petri net given to SimIUT. This gives us a possibility to rapidly experiment without truly implementing the IUT. Also making mutants is easy with such a tool available, as, for example, one may disable or add some transitions easily to achieve a slightly different simulated implementation. This is useful when analyzing the capabilities of the testing tool.

In this chapter we will concentrate on the implementation of the Bomotest tool. SimIUT and Reachability analyzer are based on the same code on the low level and therefore their implementational details can be mostly understood on the basis of Bomotest. Bomotest is based on the algorithms developed in Chapter 5. Here our objective is only to fill in the implementational perspective. To understand this chapter you should be already familiar with the algorithms of Chapter 5.

6.1 Implementation in General

The implementation has been made with C++ programming language [36]. The implementation uses some object oriented features of the language, although it has many algorithmic problems, where object oriented programming offers no advantage. The implementation consists of about 8500 lines of code.

The implementation is divided to several modules. These are the input format module, the specification module, the main program and the heuristics based coverage module.

Input Format Module
The input format module is for reading the 1-safe Petri net specification in PEP [16]-format. The format used is an older version (FORMAT_N) containing a header, place descriptions, transition descriptions, transition to places arcs and places to transition arcs in this order. The minimal net with no transitions or places goes as follows

```
PEP
PTNet
```

```
FORMAT_N
PL
TR
TP
PT
```

An example of a place definition is

```
55"PL_control___7b0_2cengaged_7d"0@0k1
```

55 is place number, `PL_control___7b0_2cengaged_7d` the place name and 0@0k1 is optional information, which is not used by Bomotest. The initial marking is given in place definitions. You can add token to place in initial marking by appending M1 to the place definition.

Transitions are specified after TR line using lines like

```
1"_o_join-udpXconfX0XpidX1XfromX0"0@0
```

Where 1 is transition number, and `"_o_join-udpXconfX0XpidX1XfromX0"` is the name of the transition. Rest of the line is not used by the Bomotest-tool. Transitions with prefix ``` ``_o_'' ``` are output transitions, those with prefix ``` ``_i_'' ``` are input transitions and transitions with neither of these prefixes are internal transitions. The semantic name of an output or an input is given after the prefix, for example ``` ``_i_ack'' ``` defines an input transition with semantic name ack.

The arcs are defined in sections after TP and PT keywords. They contain transitions to places arcs and places to transitions arcs, respectively. Definition of an arc from transition 1 to place 11 is as follows in TP section

```
1<11
```

And for an arc from place 108 to transition 9 we use following syntax in PT section

```
108<9
```

The Specification Module

The specification module provides functionality to maintain super states, traverse specification and obtain out sets and enabled inputs. The key question is to implement super states. They are set of states, so we need some way to store a set. Intensively used operations are to check whether a particular state already exists in the set and adding a state to the set.

There are several data structures which enable these operations. For example linked list is simple but inefficient alternative. Also many kinds of trees offer alternative, but we chose to use a hash table. Hash table offers amortized constant time addition and check of existence [8]. With trees these operations are likely to take time proportional to the logarithm of the amount of elements in the tree.

Therefore we have implemented a hash table to hold the markings in the current super state. We have collision resolution by chaining, so elements having an equal key are stored in linked list at the hash table position with key.

The size of the hash table is determined dynamically. The initial hash table size is 1024 slots and it is grown dynamically every time the current size is less than half the number of items in the hash table. At this point, the size of the hash table is doubled. Exponential growth of the amount of hash table slots quickly reduces the need for rehashing.

StateSetHash is an implementation of the hash table containing the states. It contains two important routines. First is to obtain all transitions enabled in the super state and detecting quiescence. This corresponds to operation $out(S)$, when the we select only output transitions and add δ when the state is quiescent. Second one is to fire transitions. Firing transitions with label a corresponds to S **after** a.

The hash tables almost never get smaller, only the situation when this happens is calculating S **after** δ. Otherwise when the **after** operation is used on regular labels, a new hash table is created for the resulting super state. This combines neatly with our rehashing strategy, as the situation having large hash table with only a small number of elements is unlikely.

Besides StateSetHash the implementation of specification modules includes classes DecodedState and EncodedState modeling markings. Intention is to model compact markings with no operations enabled with EncodedState and with DecodedState marking-net-pair having operations enabled. Operations of DecodedState are retrieving enabled transitions and firing them as well obtaining the resulting markings.

StateSetHash is also linked to coverage based heuristics by enabling registration of heuristics to the StateSetHash and informing the registered heuristics about fired transitions as well as replying to their queries about the current super state. StateSetHash uses CoverageCriteriaSet, which is built according a singleton pattern [14], containing all the registered heuristics. A heuristic is represented by the interface CoverageCriteria, which enables the heuristic to obtain information about fired transitions and affect the decisions made. Each coverage criterion in CoverageCriteriaSet has a priority. These enable use of multiple coverage criteria in fashion that first criteria with higher priorities are achieved and then the achieving lower priority criteria is targeted.

Performance of the specification module

As you might have noticed, the Algorithm 5.14 has to calculate the **after** operation after each successful input, output or timeout from the implementation. To calculate this, basically we have to check each marking in the super state and check whether any transition with the specified label is enabled. Therefore implementation of this routine is essential when considering the overall performance of the Bomotest-tool. Fortunately, not all specifications give arise to large super states causing excessive computations.

To implement the **after** -operation we have to loop over all the markings in the current super state. This is implementing by looping over all the slots in the hash table. If the hash table contains a lot of empty slots, this is ineffective. However, if there is small number of suspensions when compared to other events then we know by previous that most of the time the hash table has of size n has $\max(1023, \frac{n}{2})$ empty elements.

For each marking, we have to check all the transitions which are enabled and have the given label. This requires us to loop over all the transitions with the enabled label. We have implemented labels and label types (input, output or internal) as integers to speed up this operation. Using strings causes excessive overhead, as comparing them consumes most of the CPU time. Another optimization made is that first we compare whether the transition has the given label. This requires only two checks of integers, which are directly addressable. Only if it matches given label, we check whether it is enabled. This requires us to traverse through the net structure checking that all pre-places of the transition have a token in the current marking.

The Main Program and Communication with the IUT

The main program is basically the Algorithm 5.14 and the heuristic presented in Chapter 5 implemented in the C++ programming language. It accepts several command line options. The options enable user to choose the specification, to set the adapter used to communicate with the IUT, and to limit the CPU time used. Also, one is able to modify select several options affecting the heuristic. One can select BMC heuristic, greedy heuristic or random heuristic. Finally, the user may set coverage bound, maximum number of events before termination and the reset interval giving even finer grained control over the choices made by the heuristic.

The communication with implementation under test is done using Unix pipes and an *adapter* sitting at the other end of the pipe. Adapter is an IUT specific component translating Bomotest events to actual events understood by the IUT. Once invoked the Bomotest tool is given an adapter as a parameter. It executes the adapter redirecting its standard input and output file descriptors to Unix pipes which it reads and writes at the other end.

The adapter program must be prepared to accept and respond commands from the tester which it receives from the standard input. The Bomotest-tool uses messages separated by the new line mark to give commands. The messages are stimulate and response. A stimulate command has the syntax are `[x] Stimulate (y/z): input label`, where x is the current trace length, y and z are currently unused. Once stimulate command has been received the adapter is supposed to send an action corresponding to `input label` to the IUT. The response command has the syntax `[x] Response:`, where x is the current trace length. After receiving the response command, the adapter is expected to give the output of the IUT to the tester. The output may either be actual output or a quiescence, i.e., timeout. The response to this message is to write output message or `_` – denoting quiescence – to standard output and terminate the message with newline character.

This protocol makes the implementation of an adapter straightforward. You just have to wait for the commands and act upon them. Unfortunately there are still some practical implementation issues. The adapter must have some sort of timeouts to implement quiescence. Sometimes selecting such timeouts is hard because making them too long introduces delay to testing and on the other hand, making them too short introduces false quiescences and therefore false fail-verdicts. Also encoding complicated messages including data is confusing and might introduce really big specifications. Therefore often the adapter has to fill messages with some data not chosen by the tester and due this the tester is not aware of possible defects related to this data. However, throughly testing at least the control of the IUT is a good start.

6.2 Implemented Coverage Metrics

Currently we have implemented only one coverage metric: the Petri net transition coverage. We have defined formally this in Def. 4.31. Adding new coverage metrics is easy, because there is a well defined interface between the other parts of the tester and the coverage metric combined with a heuristic. Also the tester enables the use of multiple criteria simultaneously. In the Petri net transition coverage the aim is to fire all transitions at least once and select the next input in a way that the amount of transitions fired at least once is likely to increase.

To implement the Petri net transition coverage, we have a bit array which records whether transitions have been covered. At startup the Petri net coverage module is registered to the CoverageCriteriaSet and therefore it receives information whenever some transition is fired. When a transition is fired it simply marks the corresponding bit in the bit array. Implementation is very simple and straightforward. The only problem is that a redundant specification might have transitions which are never enabled and for such specifications we can never achieve the Petri net transition coverage. The Petri net transition coverage we have implemented does not check whether such transitions exist, because it is a time consuming operation as it requires reachability analysis, which is a computationally hard problem [40].

6.3 Implemented Heuristics

To increase the coverage, we have implemented heuristics. Heuristics are coverage metric dependent. Currently our implementation has the reset and termination conditions in the main program and not separated to the heuristic modules. A heuristic can only choose between which input to take or whether to ask an output. Termination of the algorithm happens when coverage reaches coverage bound or the maximum test events limit is reached. These are both parameters given by the user. The reset frequency is also a parameter to the Bomotest tool. It is given as a number of events, including input and output events, and once this number of events reaches the reset frequency parameter value the implementation is reset.

The random heuristic is not really a heuristic. It combines Algorithm 5.14 with Algorithm 5.15, which just makes a random choice between the enabled transition just like the greedy heuristic does when there are no non-covered inputs enabled.

Greedy heuristic is greedy search in the current super state. It basically looks for an enabled input or output which causes some non-covered transition to fire. With small probability it chooses a random transition instead of a greedily selected one, or if there exists no uncovered enabled transition it randomly chooses a enabled transition. Random choice here means uniform distribution between transitions. Another obvious possibility would be to have a uniform distribution between labels. This heuristic is Algorithm 5.16 with the LookAheadTestMove being random test move.

The BMC heuristic resembles greedy heuristic, but it uses the actual Algorithm 5.18 instead of random choice when there is no immediately enabled uncovered transition.

Implementing Lookahead with Bounded Model Checking

Our implementation of the Algorithm 5.18 is based on the bounded model checking (BMC) algorithm for 1-safe Petri nets described in [20], incorporating the process semantics optimization described in [19]. Bounded model checking is a recently introduced method [5] for exploring all the possible behaviors of a finite state system (such as a sequential digital circuit or a 1-safe Petri net) up to a fixed bound k. The idea is roughly the following. Given e.g., a sequential digital circuit, a (temporal) property to be verified, and a bound k, the behavior of the sequential circuit is unfolded up to k steps as a Boolean formula S and the negation of the property to be verified is represented as a Boolean formula $\overline{R}$. The translation to Boolean formulae is done so that $S \wedge \overline{R}$ is satisfiable iff the system has a behavior violating the property of length at most k.

In our case the temporal property to be verified says that all executions of length k (of the specification) starting from a given state $s_i \in S$, where S is the current super state of the specification, are such that the last event of that execution is invisible or a covered transition. Thus a property violation is an execution of length k with a last transition which is both visible and uncovered. Tests beginning with such an execution can lead to increasing visible Petri net transition coverage.

Our implementation tries to find an execution of length k by trying values of k from 2 to 10. If no execution could be found with bound of 10, we give up, and use fully random test move selection. We try several initial states s_i for each bound k, thus increasing our chances to find a suitable execution. The maximum number of initial states randomly sampled from the super state S is picked from the sequence $(32, 16, 8, 4, 2, 1, 1, 1, 1)$, e.g., for $k = 3$ we generate 16 different BMC instances.

The implementation of BMC uses the Smodels system [32] as underlying engine to solve the bounded model checking instances. The Bomotest tool creates problem instance based on the selected initial state and length of execution. This instance is then given to the Smodels system. If Smodels finds a solution to this problem, it is returned. This equals to finding an execution σ yielding higher coverage. Bomotest takes advantage of the returned execution σ by executing the first visible action of this sequence as well as storing the rest of the execution to continue, if possible, the desired execution on the next step. Of course continuing execution is not always possible, if an output given by the implementation is not the one in the execution.

The Algorithm 5 can in our implementation also be disabled, in which case we call it a "greedy heuristic". When it is enabled, we call it a "BMC heuristic".

To learn the exact details of the Bomotest testing tool reader is suggested to take a look at the source code, which is available through the author's home page at `http://www.tcs.hut.fi/~tpyhala/`.

7 CASE STUDIES

We have experimented with the Bomotest implementation using simple synthetic experiments and larger real software based experiments. With synthetic examples we can evaluate effectiveness of the heuristics. By experimenting with real, though academic, software we wish to evaluate how effective our coverage metric is in finding real defects.

7.1 Evaluation of Heuristics with Synthetic Examples

Our synthetic examples are derived from the simple idea of having a combinatoric lock. It is a device often found in suitcases, where one has to select the correct numbers to open the lock. With this experiment we wish to verify that our heuristic is able to efficiently increase the coverage. It was also interesting to find out how efficiently Bomotest detects the faulty implementations.

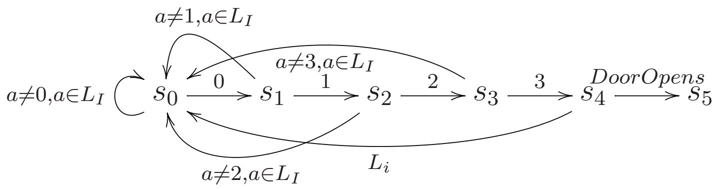

FIGURE 18: Combinatoric lock with code 0123 as a LTS

Test Setting

In the first combinatoric lock experiment we ran the tests using SimIUT simulating the correct implementation according the specification. In this experiment we did not target finding faults, but increasing coverage efficiently. Therefore we did not use faulty implementations but SimIUT with correct specification instead. As the result we observed how fast the coverage rose.

In the second experiment with locks we used faulty implementations to measure how efficiently Bomotest is able to detect them using different heuristics. This enables us to compare the practical efficiency of heuristics when targeting to find the bugs.

We used maximum CPU time usage of 900 seconds and no resets as options to Bomotest during testing. The tests were run on a 1GHz AMD Athlon processor running Debian GNU Linux 3.0 and Linux kernel version 2.4.18.

Results

The results of the first experiment for the simple combinatoric locks are presented in Table 1. The table contains the arithmetic average of the number of events required to obtain 100% coverage and the average of CPU seconds used during the testing for each heuristic. The average is calculated from 10 runs. The table is parametrized with the length of the lock being tested. The table contains text FAIL if all the runs did not complete before exceeding the CPU usage limit.

As the second experiment we have analyzed the capability of Bomotest to detect a faulty implementation. We used locks having incorrect sequence and tried to detect these with the Bomotest. The number of mutants detected versus the number of test events executed is plotted in Fig 19. It is desirable to detect faulty implementation with as small number of events as possible.

	Random		Greedy		BMC	
Length	CPU	Events	CPU	Events	CPU	Events
3	3.454	10456.9	0.611	1839.1	0.112	65.7
4	25.985	71819.7	6.675	18476.5	0.244	112.7
5	311.691	786563.8	75.481	188763.3	0.425	163.8
6	FAIL	FAIL	743.713	1729456.8	0.73	230.6
7	FAIL	FAIL	FAIL	FAIL	1.148	306.2
8	FAIL	FAIL	FAIL	FAIL	1.824	405.2
9	FAIL	FAIL	FAIL	FAIL	2.786	516.7
10	FAIL	FAIL	FAIL	FAIL	4.427	661

TABLE 1: Results for simple combinatoric lock, CPU being the CPU time used in seconds

The mutants used are such that the two last digits of the code have been altered. With lock of length four there are 99 possible mutants, and all of these have been run with 10 different seeds totaling 990 mutants.

A combinatoric lock is an example of a specification which is hard to cover (w.r.t. Petri net transition coverage) with purely random heuristic. Time and events required to cover the simple combinatoric lock grow exponentially. Another observation which can be made is that greedy heuristic is able to achieve the coverage for one step longer locks within the specified time bound and uses around one fourth of the events to achieve the coverage on equal length lock. Therefore we may conclude that greedy heuristic is more effective than random, although it does change the amount of required events only by a constant factor and the growth rate of CPU time seems to be exponential still. Especially, if you look at the number of mutants detected with certain number of executed test events the greedy heuristic does not differ much from the random heuristic.

The BMC based heuristic performs exceptionally well in this case. However, the lookahead looks 10 steps forward and therefore the performance is likely to get worse with locks containing a longer code. Fortunately, up to this point we obtained exceptionally good results with the BMC heuristic. If you look at the number of mutants detected versus the number of test events used you can notice that BMC is performing very effectively when compared to greedy and random in this case.

7.2 Evaluation of the Test Selection Method

To evaluate the coverage metric, we wish to find out whether it is successful in finding errors. For this, we need an appropriate case, i.e., some system which we may test with our Bomotest-tool. Tretmans et al have used conference protocol implementation [11] to test their testing tools [9]. We decided to use the same implementation, as it is already available, and we wish to compare our results to those of Tretmans et al.

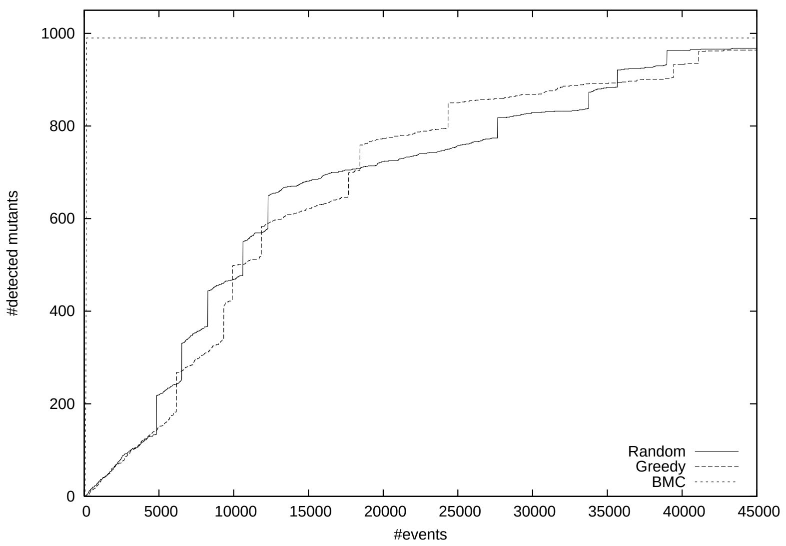

FIGURE 19: Cumulative number of detected mutants versus number of testing events in Combinatoric lock experiment.

Conference Protocol

The conference protocol implements a chat service. It enables users to form conferences and exchange messages with other partners in the conference. [11]

A conference service access point (abbreviated CSAP) offers the service primitives: join(nickname, conference), datareq(message), dataind(nickname, message) and leave with no parameters. At first a user has to join a conference. Once an active member of a conference, an user is able to send messages to other members using datareq and receive such messages sent by others using dataind. Once finished conferencing, the user may issue leave() primitive.

A Conference protocol entity (abbreviated CPE) uses UDP as the lower level service to implement the conference service. Each service primitive maps as one or more received or sent UDP PDU. The CPE has two sets, a preconfigured set of all potential conference partners and a dynamic set of current conference partners.

Issuing a join primitive causes CPE to send all the potential partners an UDP PDU, called join-PDU, containing who is joining to which conference. Existing members of the conference then add the joiner to their dynamic member sets and reply with an answer PDU, called answer-PDU. Receiving answer-PDU causes corresponding CPE add the sender of the PDU to their conference partner set. Once datareq primitive is issued, CPE sends data-PDU to all partners in the set of current conference partners. Receiving data-PDU causes a dataind primitive to be issued in respective CPE. Receiving leave-PDU causes respective CPE to remove the sender from conference member set. Sending the leave-PDU is caused by leave primitive at CSAP, and it is sent to all the members of the current conference.

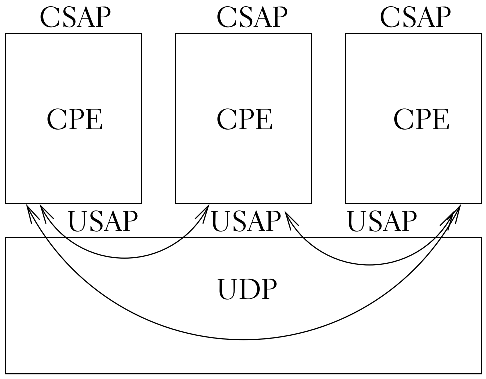

FIGURE 20: The conference protocol with 3 CPEs

Test Setting

We have run the experiments with the conference protocol using the Bomotest tool. Apart from the Bomotest tool and conference protocol implementation, we have a Petri net specification of the conference protocol and an adapter.

The specification has been developed as a high level net [33], which can be converted to a 1-safe Petri net. A high level net is more expressive than 1-safe Petri net, and hence we are able to have a smaller and more understandable specification describing exactly the same I/O behavior. In the conversion to labelled 1-safe Petri net we have used Maria [25] analyzer and custom made Perl scripts.

Basically we have modeled the situation as in Fig. 21. The IUT is a CPE, and we have also two virtual CPEs. The PCOs are the USAPs of the virtual implementations and the CSAP of the IUT. The adapter communicates with the CPE using Unix pipes and it receives and sends messages of the virtual CPEs using UDP sockets.

The specification contains no buffering for inputs from the virtual CPEs. It models messages from virtual partners and from CSAP as being received immediately. Output through UDP contains such buffering, that the PDU sent is put to a place called "Internet", which is able to contain at maximum of 1 PDU of each (receiver, sender, type)-combination. Once a packet is in the place "Internet", it may be received by a virtual CPE at any moment through transition ReceiveVirtual. The full high-level specification is available through the web.

The conversion process has been made using Maria and Perl. Maria is able to translate the high-level net to a Petri-net in input format of the PEP-tool [16]. The resulting net is 1-safe, if the high-level net has been carefully designed to be such, which we have done. We also used the reduction option M of the Maria while unfolding the net. Once in the PEP-format, we used a Perl script to mark the transitions as visible or internal. This process resulted a 1-safe Petri net specification suitable to be used with Bomotest.

Besides the specification the Bomotest requires the adapter to convert the messages suitable for the IUT. Adapter is implemented in very straightforward way. Each message from Bomotest can be directly translated to a message in PCOs - the UDP sockets or the pipe. If the Bomotest requests for an output, adapter will check if some of the PCOs has an input available. This

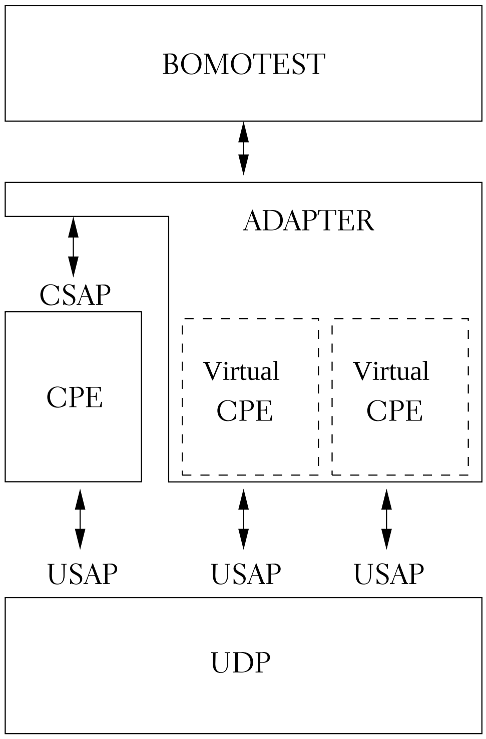

FIGURE 21: An Overview of the Test Setup

is done non-deterministically using the Unix *select* [35] system-call. To give implementation possibility respond to our stimuli, we used a timeout of one second before the response was interpreted as suspension.

To reset an implementation, we have used a strategy where we kill the existing CPE instance using Unix signals. Then we flush all the sockets and wait for few seconds to any leftover messages to arrive. After this we restart the implementation and continue.

Testing was carried out in the Linux environment. We measured combined number of events at all PCO's. These events are inputs, outputs and resets. We assumed that major cost (cost equals to time here) in testing would be executing an event at PCO. This seems to be the case in our case too because of the timeouts mentioned. Of course some events are more expensive than others. Reset and suspension especially take some time.

The CPE used contains intentional errors, which can be enabled with command-line switch -mutant. We have tested a supposedly conforming implementation of the CPE and also intentionally mutated instances of it. In mutant column there is mutant code, or original for the original conforming implementation. Tests have been run until Bomotest-tool has reached 100% coverage, or detected a non-conforming implementation.

The results in Table 2 have been obtained using no resets at all, averages correspond to 10 runs with different pseudo random generator seeds, and the average is the arithmetic average. Ratio gives the ratio between heuristic 2 (BMC) testing events and heuristic 0 (random) testing events. The pseudo random generator used is the BSD random function (random()) found in the libc-library of Debian GNU/Linux system version 2.1. See the URL provided in the end of chapter for more information.

Figure 22 shows graphically how fastly mutants are detected with each heuristic. In x-axis we have the number of test events executed and in the y-axis we have the number of mutants detected with less or equal than this amount of executed testing events.

Results

The results in Table 2 show that in all cases except one the BMC heuristic was better on average than random heuristic. However in one case BMC heuristic was slightly worse. The BMC heuristic found all the faulty implementations, although we had to set maximum testing events and not use 100% transition coverage as a termination condition. Therefore we may conclude that 100% transition coverage is not sufficient condition to detect these mutants.

We have not analyzed in detail why 100% transition coverage is not sufficient to detect these mutants. However, we see that transition coverage is a very high level coverage metric in the sense of hierarchy presented in Chapter 4 and as such it is able to only guide testing at a very high level and therefore we need randomness to gain the traces leading to errors. Only firing all the transitions once does not induce enough testing.

During testing, we had also a faulty implementation of the adapter, which did never give the data indication at IUT CSAP message. This kind of a fault was efficiently detected with greedy heuristic and Petri net transition coverage. It is a fault associated with one transition only, and therefore we assume that our heuristics and coverage metric would be very efficient against such faults.

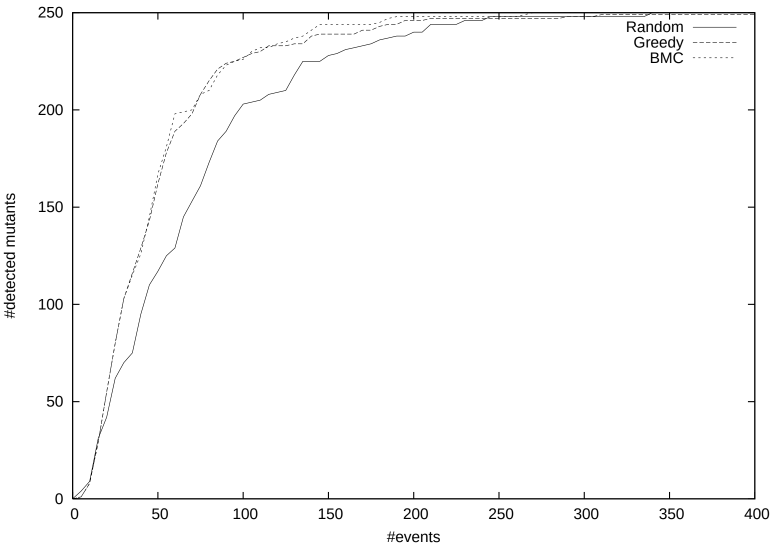

FIGURE 22: Conference protocol: Cumulative number of detected mutants versus testing events

The Fig 22 shows that most of the time BMC heuristic is able to detect more mutants with test effort limited to n events, the same happens to greedy heuristic. It is interesting that we see almost no benefit over greedy heuristic when using BMC.

In this example we have already noticed that firing all the transitions is not sufficient to detect all the mutants. We can see that avoiding redundant work is advantageous, but there is also advantage from randomly traversing the specification. Latter detects bugs, whose detection is not implied by fulfilling our coverage criteria. Specification is also connected in a manner, that there are no special sequences of events leading to a totally new area in specification. Compare this to the combinatorial lock, where such sequence exists. Bugs have neither been distributed evenly around the specification. Actually it is not required to test everything to detect small set of bugs.

We speculate that performance of the greedy heuristic is related to the fact that it avoids some of the redundant work while maintaining higher amount (than BMC) of random walking around the specification. First is disadvantage and latter is advantage when compared to the BMC heuristic. As total effect the performance of it is almost the same as the BMC's.

For further information on the experiments found in this chapter, please see the web site at the address:
`http://www.tcs.hut.fi/~tpyhala/experiments/mscthesis/`. It contains specifications for the experiments.

8 CONCLUSIONS

Automated testing tools are an emerging technology. Best of the tools will contain automated test selection methods to improve the quality of testing. In this work we have formally defined a framework for specification coverage aided test selection. This includes a precise definition of some coverage metrics as well as a refined algorithm of [41] for implementing test selection taking advantage of the metrics. As an interesting implementation technique we have used bounded model checking (BMC) for searching executions increasing the coverage. Finally an on-the-fly testing tool with specification coverage based test selection methods was implemented.

The contribution also includes experiments with the implemented tool, which is called Bomotest. We can conclude from the experiments that in some situations such test selection method is able to increase the performance by several orders of magnitude, however, in some other situations we see only little difference. Further analyzing where these methods are most useful and why is left for future work.

However, we believe that in this work we have given the testing community more experiences on test selection with a new method. In future we see that it is possible to improve the presented ideas and try to apply them to more realistic case studies. The conference protocol case study is, although a good starting point, a little bit academic, and quite a small one. Also the errors in the implementation are artificial, not real. Having more case studies would also give better insight whether the ideas and the methods have use in practice. The coverage metric used is simple and therefore it can be improved. For example, implementing the LTS coverage presented in Chapter 4 would give more granularity, although it might be impossible for large specifications due to larger memory consumption needed to store the coverage information.

Mutant	Random	Greedy	BMC	$\frac{\text{BMC}}{\text{Random}}$
467	74.5	48.8	36.5	0.49
687	68.8	39.7	36.3	0.52
293	80.0	48.4	43.8	0.54
856	97.3	70.6	53.8	0.55
214	103.3	51.3	58.7	0.56
777	103.3	51.3	58.7	0.56
332	122.6	74.8	71.8	0.58
348	93.6	74.1	59.0	0.63
294	154.4	156.4	105.7	0.68
111	38.9	29.2	27.4	0.70
247	38.8	29.2	27.2	0.70
674	15.1	10.6	10.6	0.70
345	55.2	38.9	40.2	0.72
945	45.2	34.0	32.7	0.72
749	20.3	15.2	15.2	0.74
358	45.5	35.7	34.4	0.75
289	73.1	52.3	56.1	0.76
276	23.2	18.0	18.0	0.77
384	47.8	43.2	39.0	0.81
548	75.3	88.9	62.8	0.83
462	84.7	59.5	71.8	0.84
738	84.7	59.5	71.8	0.84
100	42.1	38.8	41.2	0.97
836	42.1	38.8	41.2	0.97
398	94.7	79.4	103.6	1.09

Table 2: Number of events required to detect faults in the conference protocol experiment

Of course, there are things to improve in the heuristics, too. We have assumed that implementation is co-operating with us in the sense that it lets us to uncovered areas of specification. We can also consider testing as a two player game. If the implementation plays against us, we could try to build strategies which would enable the tester to achieve the coverage metric used also while the implementation plays the best possible strategy to prevent us from achieving the coverage. We see that this could be similar in spirit to min-max algorithms used when designing artificial intelligence for games like chess.

We also see that to be able to compare the heuristics and coverage metrics a definition of some kind framework to enable these comparisons is needed. Carefully defined fault models could prove to be such a framework. Therefore it would be interesting to see the theory behind fault models to mature. Concepts like completeness and soundness are far too rough tools to analyze heuristics, coverage metrics and algorithms in detail. We need more refined tools for analyzing fault detection capabilities.

We have not analyzed in detail the connection between increased fault detection capability of testers and increased coverage. Testing is in practice only sound in the sense that it is able to detect bugs, but not able to prove the program correct. On the other hand, for example sending inputs to the IUT may increase the coverage, although it is impossible to detect bugs by sending inputs. The faults are detected by observing outputs. The problem is: what is the efficient way to detect bugs?

ACKNOWLEDGMENTS

This is the report version of my Master's Thesis. The report was done in the Laboratory for Theoretical Computer Science at HUT while I was a research assistant. I would like to thank my instructor D.Sc. (Tech.) Keijo Heljanko, M.Sc. (Tech.) Antti Huima from Conformiq Oy Ltd. and supervisor Prof. Niemelä for patience, guidance, discussions and comments on draft versions. I'm also thankful for the financial support for the research behind this report, namely Conformiq Oy Ltd. and Academy of Finland (Project no. 53695).

Also many others from the laboratory staff have been very helpful and nice to me and I've had great time working with you.

As last, but not least, I would like to thank my parents and siblings as well as my friends for support in non-technical matters.

REFERENCES

[1] P.E. Ammann and P.E. Black. A specification-based coverage metric to evaluate test sets. *International Journal of Reliability, Quality and Safety Engineering*, 8(4):275–300, December 2001.

[2] P.E Ammann, P.E. Black, and W. Majurski. Using model checking to generate tests from specifications. In M.G. Staples, J. Hinchey and S. Liu, editors, *2nd IEEE International Conference on Formal Engin-*

eering Methods (ICFEM'98), pages 46–54, Brisbane, Australia, 1998. IEEE Computer Society.

[3] B. Beizer. *Software testing techniques - 2nd ed.* International Thomson Computer Press, The United States of America, 2001.

[4] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment, 1999.

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, pages 193–207. Springer, March 1999.

[6] G.V. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault models in testing. In Kroon J., Heijink R.J., and Brinksma E., editors, *Protocol Test Systems IV*, volume C-3 of *IFIP Transactions*, pages 17–30. North-Holland, 1992.

[7] R. Castanet and D. Rouillard. Generate certified test cases by combining theorem proving and reachability analysis. In *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems*, pages 267–282, Berlin, Germany, March 2002.

[8] T.H. Cormen, C.E Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, USA, 1990.

[9] R.G. de Vries and J. Tretmans. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer*, 2(4):382–393, March 2000.

[10] J. Desel and W. Reisig. Place/transition Petri nets. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:122–173, 1998.

[11] J. Feenstra. Conference protocol case study. WWW pages at address `http://www.cs.utwente.nl/ConfCase`, last updated 1999-11-18.

[12] L. M. G. Feijs, N. Goga, S. Mauw, and J. Tretmans. Test selection, trace distance and heuristics. In *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems*, pages 267–282, Berlin, Germany, March 2002.

[13] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In A. Alur and T. Henzinger, editors, *Conference on Computer-Aided Verification (CAV '96), New Brunswick, New Jersey, USA*, LNCS 1102. Springer-Verlag, July 1996.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[15] N. Goga. An optimization of the TorX test generation algorithm. In *SAM2000 - 2nd Workshop on SDL and MSC*, pages 173–188, Col de Porte, Grenoble, June 2000. VERIMAG, IRISA, SDL Forum Society.

[16] B. Grahlmann. The PEP tool. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 440–443. Springer-Verlag, 1997.

[17] A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 12–21, July 2002.

[18] J.A Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[19] K. Heljanko. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR'2001)*, volume 2154 of *Lecture Notes in Computer Science*, pages 218–232, Aalborg, Denmark, August 2001. Springer.

[20] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'2001)*, volume 2173 of *Lecture Notes in Artificial Intelligence*, pages 200–212, Vienna, Austria, September 2001. Springer.

[21] J. Helovuo and S Leppänen. Exploration testing. In *2nd IEEE International Conference on Application of Concurrency to System Design (ICACSD 2001)*, pages 201–210, Newcastle upon Tyne, U.K., June 2001. IEEE Computer Society.

[22] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[23] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[24] International Telecommunication Union, Geneva, Switzerland. *CCITT Specification and Description Language (SDL), recommendation z.100*, October 1996.

[25] M. Mäkelä. Maria: modular reachability analyser for algebraic system nets. In Javier Esparza and Charles Lakos, editors, *Application and Theory of Petri Nets 2002, 23rd International Conference, ICATPN 2002*, volume 2360 of *Lecture Notes in Computer Science*, pages 434–444, Berlin, Germany, 2002. Springer-Verlag.

[26] U.M. Maurer. A universal statistical test for random bit generators. *Lecture Notes in Computer Science*, 537, 1991.

[27] K.L. McMillan. Symbolic model checking. Kluwer Academic Publishers, 1993.

[28] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, USA, 1995.

[29] C.H. Papadimitriou and Harry R. Lewis. *Elements of the theory of computation*. Prentice-Hall, 2nd edition, 1998.

[30] A. Pretschner. Classical search strategies for test case generation with constraint logic programming. In *Proceedings of Formal Approaches to Testing of Software (FATES'01)*, pages 47–60, Aalborg, Denmark, August 2001.

[31] A. Rautiainen. Conformance testing in distributed testing architecture. Master's thesis, Helsinki University of Technology, Espoo, Finland, 1995.

[32] Patrik Simons. Extending and implementing the stable model semantics. Research Report A58, Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, Espoo, Finland, April 2000. Doctoral dissertation.

[33] E. Smith. Principles of high-level net theory. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:174–210, 1998.

[34] I. Sommerville. *Software engineering - 6th ed.* Addison Wesley, Pearson Education Limited, Edinburgh Gate, Harlow, Essex CM20 2JE, England, 2001.

[35] W. R. Stevens. *UNIX Network Programming*. Prentice Hall, USA, 1990.

[36] B. Stroustroup. *The C++ Programming Language – 3rd ed.* Addison Wesley, USA, 1997.

[37] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE Conference on Automated Software Engineering (ASE)*, 2001, October 1998.

[38] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[39] J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[40] A. Valmari. The state space explosion problem. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:429–528, 1998.

[41] R.G. de Vries and J. Tretmans. On-the-Fly Conformance Testing using SPIN. In G. Holzmann, E. Najm, and A. Serhrouchni, editors, *Fourth Workshop on Automata Theoretic Verification with the* SPIN *Model Checker*, ENST 98 S 002, pages 115–128, Paris, France, November 2, 1998. Ecole Nationale Supérieure des Télécommunications.

[42] S. T. Vuong and J. Alilociv-Curgus. On test coverage metrics for communication protocols. In Jan Kroon, Rudolf Jan Heijink, and Ed Brinksma, editors, *Protocol Test Systems IV*, volume C-3 of *IFIP Transactions*, pages 31–45. North-Holland, 1992.

[43] J.A. Whittaker. What is software testing? And why it is so hard? *IEEE Software*, 17(1):70–79, 2000.

[44] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *Journal on Software Testing Verification and Reliability*, 10(4):229–248, 2000.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A80	Tommi Junttila On the Symmetry Reduction Method for Petri Nets and Similar Formalisms. September 2003.
HUT-TCS-A81	Marko Mäkelä Efficient Computer-Aided Verification of Parallel and Distributed Software Systems. November 2003.
HUT-TCS-A82	Tomi Janhunen Translatability and Intranslatability Results for Certain Classes of Logic Programs. November 2003.
HUT-TCS-A83	Heikki Tauriainen On Translating Linear Temporal Logic into Alternating and Nondeterministic Automata. December 2003.
HUT-TCS-A84	Johan Wallén On the Differential and Linear Properties of Addition. December 2003.
HUT-TCS-A85	Emilia Oikarinen Testing the Equivalence of Disjunctive Logic Programs. December 2003.
HUT-TCS-A86	Tommi Syrjänen Logic Programming with Cardinality Constraints. December 2003.
HUT-TCS-A87	Harri Haanpää, Patric R. J. Östergård Sets in Abelian Groups with Distinct Sums of Pairs. February 2004.
HUT-TCS-A88	Harri Haanpää Minimum Sum and Difference Covers of Abelian Groups. February 2004.
HUT-TCS-A89	Harri Haanpää Constructing Certain Combinatorial Structures by Computational Methods. February 2004.
HUT-TCS-A90	Matti Järvisalo Proof Complexity of Cut-Based Tableaux for Boolean Circuit Satisfiability Checking. March 2004.
HUT-TCS-A91	Mikko Särelä Measuring the Effects of Mobility on Reactive Ad Hoc Routing Protocols. May 2004.
HUT-TCS-A92	Timo Latvala, Armin Biere, Keijo Heljanko, Tommi Junttila Simple Bounded LTL Model Checking. July 2004.
HUT-TCS-A93	Tuomo Pyhälä Specification-Based Test Selection in Formal Conformance Testing. August 2004.

ISBN 951-22-7240-7
ISSN 1457-7615